Pontifícia Universidade Católica
do Rio de Janeiro

**Rodrigo Costa Mesquita Santos**

# A GALS Approach for Programming Distributed Interactive Multimedia Applications

**Tese de Doutorado**

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Noemi de La Rocque Rodriguez

Rio de Janeiro
October 2018

Pontifícia Universidade Católica
do Rio de Janeiro

**Rodrigo Costa Mesquita Santos**

# A GALS Approach for Programming Distributed Interactive Multimedia Applications

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the undersigned Examination Committee.

**Prof. Noemi de La Rocque Rodriguez**
Advisor
Departamento de Informática – PUC-Rio

**Prof. Francisco Figueiredo Goytacaz Sant'Anna**
– UERJ

**Prof. Roberto Ierusalimschy**
Departamento de Informática – PUC-Rio

**Prof. Sérgio Colcher**
Departamento de Informática – PUC-Rio

**Renato Fontoura de Gusmão Cerqueira**
IBM Research – IBM

**Prof. Marcelo Ferreira Moreno**
– UFJF

Rio de Janeiro, October 4th, 2018

**Rodrigo Costa Mesquita Santos**

Rodrigo has Bachelor's Degree (2010) and Master's Degree (2013) in Computer Science from Federal University of Maranhão, where he worked as a researcher for the Laboratory of Advanced Web Systems (LAWS Lab). In 2013, he joined the TeleMídia Lab. of PUC-Rio, working on the evolution and maintenance of the Ginga-NCL middleware and related tools. From 2016 to 2018, he was a Research Software Engineering intern at IBM Research Brazil. Currently, Rodrigo is a Research Software Engineer at IBM Research Brazil, where he works mainly with knowledge engineering and AI.

Bibliographic data

Santos, Rodrigo Costa Mesquita

A GALS Approach for Programming Distributed Interactive Multimedia Applications / Rodrigo Costa Mesquita Santos; advisor: Noemi de La Rocque Rodriguez. – 2018.

v., 143 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2018.

Inclui bibliografia

1. Informática – Teses. 2. Céu;. 3. Multimedia;. 4. Multi-dispositivos;. 5. Determinismo;. 6. Consistência. I. Rodriguez, Noemi de la Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my mother, to my grandmother (*in memoriam*), to Prof. Luiz Fernando (*in memoriam*) and to the love of my life (Adriana Rabelo).

## Acknowledgements

First of all I would like to thank God for all the strengths He gave me throughout this journey, especially when I thought I would not be able to complete it. Without His blessing I would not get here.

I'm very grateful to my Mom who always supported me since my birth. She always taught me that education is one of the cleanest and shortest paths to success (even though I don't necessarily agree that it is that short). She gave me the opportunity to focus only on my studies for the most part of my academic journey. I'm sure that completing this doctorate is a dream that comes true more to her than from me. Thank you, Mom, for always been there for me. I will always be grateful to you.

I want to say a big THANK YOU for all the support that the love of my life, Adriana Rabelo, gave to me throughout this time. I also want to apologize for the dozens (maybe hundreds) of nights and weekends that this work stole from us. With her, I shared all the joys and difficulties to complete this thesis, and I always found in her a safe haven. She is one of the most important person in my life and I truly love her without restrictions. I hope to be with you for the rest of my life.

I'm grateful to Prof. Luiz Fernando (*in memoriam*) who accepted me as his student before knowing me well when I joined the TeleMídia Lab. With him, I learned so much and I'm very proud of being part of the select group of his students. Unfortunately, he is not here to celebrate with me, but I'm sure he would be very happy if he was. Thank you, Professor. NTPPM.

My advisor, Noemi, deserves a huge thank you. She accepted the challenge of advising a student in the middle of his doctorate and from a completely different field of study. And surprisingly that worked very well for us (at least for me, I hope she shares the same feeling). We had some troubles at the beginning due to our different backgrounds, but we got through that and this is the result of our work together. And this would not be possible without the help of Francisco, who was invited to help in this work and promptly accepted it. I've learned a lot with them and I'm confident that I made a good choice when I asked them to advise me. Thank you both for all the guidance and patience.

A doctorate is supposed to be an individual effort, but this work has significant contributions from my bros. I want to say thank you for the members of the TeleMídia Lab (even those honorary—Marcos Roriz) for all the valuable discussions. Assuming the risk of be unfair, I want to nominally thanks Alan, Álvaro, André, Busson, Eduardo, Guilherme, Rafael (Juba) and Roberto.

I cannot forget to cite the brothers that my academic journey gave me: Alex, José Rios, Thiago and Wallas. You guys are more important to me than you think you are.

There are two Professors that directly influenced me to pursue the doctorate degree: Carlos de Salles and Mario Meireles. They accepted me as an undergraduate student in their lab (which wasn't exactly a lab back then) and taught me what is and the importance of the research. Thank you for instigating me to become a researcher. You two have fundamental importance in my formation and I probably would not get here without your support at the beginning of my academic life.

And finally, I would like to thank CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) for funding this work.

## Abstract

Santos, Rodrigo Costa Mesquita; Rodriguez, Noemi de la Rocque (Advisor). **A GALS Approach for Programming Distributed Interactive Multimedia Applications**. Rio de Janeiro, 2018. 143p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In this, work we investigate how to guarantee two properties in the development of interactive distributed multimedia applications: determinism and consistency. Determinism is a property of individual nodes in a distributed application and states that a program always produces the same output when fed with the same input. Consistency is a property of the whole system and states that all nodes should have the same view of the order of events. We evaluate the use of the synchronous language Céu in the context of multimedia programming for guaranteeing the determinism property. Regarding consistency, we evaluate the GALS (Globally Asynchronous Locally Synchronous) architecture for enforcing consistency. Traditionally, multimedia applications are developed using either a domain specific language or a general purpose language supported by specialized frameworks. Neither of the two approaches promotes the development of deterministic and consistent interactive distributed multimedia applications. Our investigation of the use of synchronous languages in the multimedia field led to the development of Céu-Media, a deterministic multimedia library for the synchronous language Céu, and Mars, a GALS middleware for interactive distributed multimedia applications. The results of this thesis indicate that using the guarantees of the synchronous language Céu it is possible to develop deterministic multimedia applications using Céu-Media. Furthermore, they also indicate that the consistency model enforced by the GALS middleware Mars guarantees that all nodes always agree upon the order of events in a distributed presentation. We validate our proposal by discussing the development of real-world distributed multimedia applications proposed by the research community using both, Céu-Media and Mars, highlighting the main advantages and also the drawbacks of using our approach.

## Keywords

Céu; Multimedia; Multi-device; Determinism; Consistency

## Resumo

Santos, Rodrigo Costa Mesquita; Rodriguez, Noemi de la Rocque. **Uma Abordagem GALS para a Programação de Aplicações Interativas Multumídia Distribuídas**. Rio de Janeiro, 2018. 143p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho, investigamos como garantir duas propriedades no desenvolvimento de aplicações multimídia distribuídas interativas: determinismo e consistência. Determinismo é uma propriedade individual dos nós em uma aplicação distribuída e refere-se à característica de um programa sempre produzir a mesma saída a partir de uma mesma entrada. Consistência é uma propriedade de todo o sistema e está relacionada a todos os nós terem sempre a mesma visão da ordem dos eventos. Avaliamos o uso da linguagem síncrona Céu no contexto de programação multimídia para garantir o determinismo. Em relação à consistência, avaliamos se a arquitetura GALS (*Globally Asynchronous Local Synchronous*) é capaz de prover consistência. Tradicionalmente, aplicações multimídia são desenvolvidas usando linguagens de domínio específico ou linguagens de propósito geral utilizando frameworks especializados. Nenhuma dessas duas abordagens promove o desenvolvimento de aplicações multimídia distribuídas interativas determinísticas e consistentes. Nossa investigação sobre o uso de linguagens síncronas no campo de multimídia levou ao desenvolvimento de Céu-Media, uma biblioteca multimídia determinística para a linguagem síncrona Céu, e Mars, um middleware GALS para aplicações multimídia distribuídas interativas. Os resultados desta tese indicam que, usando as garantias da linguagem síncrona Céu, é possível desenvolver aplicações multimídia determinísticas usando Céu-Media. Além disso, eles também indicam que o modelo de consistência implementado pelo middleware GALS Mars garante que todos os nós sempre concordem com a ordem dos eventos em uma apresentação distribuída. Nós validamos nossa proposta discutindo o desenvolvimento de aplicações multimídia distribuídas propostas pela comunidade de pesquisa usando Céu-Media e Mars, destacando as principais vantagens e também as desvantagens em usar nossa abordagem.

## Palavras-chave

Céu; Multimedia; Multi-dispositivos; Determinismo; Consistência

# Table of contents

# List of figures

# List of tables

## List of codes

# 1
# Introduction

The proliferation of personal multimedia-enabled devices, such as smartphones, tablets, smartwatches, etc., has encouraged the development of multimedia applications running on multiple devices, known as *distributed* or *multi-device* multimedia applications. The term *distributed multimedia applications* is overloaded in literature. Some works use it to refer to applications in which the *multimedia content* is distributed among different servers, regardless of whether the presentation occurs in single or multiple devices [5, 6, 7]. Others use this very same term to describe applications whose *presentation* is designed to be executed across multiple devices [8, 9]. In this work, we use distributed multimedia applications with this second meaning. We use the terms *multi-device* and *distributed (multimedia) applications* interchangeably.

There are different types of multi-device applications. Here we are interested in applications in which each device complements one another, creating an experience as a connected group. Following Levin's terminology [10], this class of applications is called *complementary*. Consider the following scenario as an example of these applications: Alice teaches an online course. Slides and videos, controlled by Alice, are presented on her device and on all students' devices. At some point, one of the students has a question related to the class. Alice temporarily gives the student access to control the video in all connected devices, and the student rewinds it to explain the origin of his question. Afterwards, Alice withdraws the control from the student and continues.

Complementary applications involve users interaction with multiple devices at the same time. There are two types of interactions in these applications: collaboration-based and control-based [10]. Devices working together to achieve a goal characterizes the former type. A device partially controlling an application running on another device is an example of the latter.

For the collaboration and/or control be effective, all devices should have the same view of the whole system. However, the lack of an accurately synchronized global clock and asynchronous user interactions can hinder that [11]. In the example above, let's suppose that multiple students concurrently request control over the video. Depending on how the system coordinates the response to those interactions, devices may reach inconsistent states. For instance, con-

sider that the system should grant control over the video to the first student that requests it. If not all devices agree upon who made the first request (a feasible assumption due to the absence of global time) multiple students may try to control the video at the same time.

The development of interactive distributed multimedia applications involves issues from at least two different fields: multimedia and distributed systems. On the one hand, one has to be familiar with low level details of how media content is stored, coded, decoded and synchronized to program the multimedia aspect of applications. On the other hand, one must tackle classical problems of distributed systems, such as clock synchronization, consistency maintenance, and distributed consensus, when programming the interactions among devices. Even though the research community has made significant progress in both areas, there is still a lack of comprehensive proposals that combine these advances into a unified programming model.

In this thesis, we aim to investigate the development of these applications from the programming perspective. Specifically, our main concern is to guarantee two properties: *determinism* and *consistency*. The former is a property of individual processes, therefore whenever we use the term *determinism* we are referring to local[1] applications. The latter is a property of the whole system, so whenever we use the term *consistency* we are referring to distributed systems.

The definition of determinism in the multimedia domain should consider the timing aspect because it can impact the synchronization of applications. We say that a multimedia application behaves deterministically if, in any execution, it always produces the same sequence of outputs, executing the same sequence of steps at the same time instants, when submitted to the same sequence of inputs. As stated by Benveniste and Gary, *"there is no reason the engineer should want his [system] to behave in some unpredictable manner"* [12]. In general, the use of deterministic languages helps programmers to better reason about their source codes, because they can precisely compute the sequence of steps that programs will execute for a given input.

Regarding the second property, there are different consistency models and definitions proposed in literature. Here we use a definition that is based on the sequential consistency model defined by Lamport [4], but with an extension to accommodate the timing aspect—we call it timing-sequential consistency model. The sequential consistency model states that a system is said consistent if *"the result of any execution is the same as if the operations of all the [processes] were executed in some sequential order, and the operations*

[1]Throughout this thesis, the term *local* is used as synonym of a stand-alone (and antonym of distributed) application and should not be confused with applications running on a LAN.

*of each individual [process] appear in this sequence in the order specified by its program"* [4]. However, for multimedia applications just the ordering of messages is not enough for a consistency model: different processes may reach completely different states if they execute the same operations at different time instants. Thus, the consistency model that we adopt adds the constraint that all operations should not only be executed in the same order, but also at the same time in all processes. In this work, the *consistency* property is defined in terms of this model.

The timing-sequential consistency model implies that: i) there is a total ordering of events on which all processes agree; ii) all messages sent from a given process are received in the same order by all others; and iii) all processes receive messages at the same time. If a system enforces this consistency model, one does not have to worry about implementing algorithms to ensure that all processes of a distributed system have the same global view.

One can develop (local or distributed) multimedia applications either using domain specific (DSLs) or general purpose languages. DSLs for multimedia, hereafter called *multimedia languages*, can be used for developing applications without directly programming low-level operations. NCL [13], SMIL [14], and HTML5 [15] are examples of such languages. An alternative to DSLs is to use general purpose languages, combined with multimedia frameworks (e.g., GStreamer, FFmpeg, Libav) following a more imperative approach. We argue that it is hard to ensure determinism and consistency using these approaches. And this is not just an implementation matter, it is because the programming models they promote were not designed to embrace these properties.

Multimedia languages, in general, do not have deterministic semantics. A direct consequence is that there is no guarantee that an arbitrary program written in these languages behaves identically in multiple executions. Usually, they also lack support for programming distributed applications. NCL and IPML (a SMIL-based language) are exceptions because of their constructs for developing multi-device programs, but they fail to guarantee consistent executions in all cases. Multimedia frameworks, on their part, are intrinsically multithreaded. Ensuring determinism when multithreading is involved is a well-known problem [16]. And these frameworks usually do not implement typical functionalities of distributed systems.

We advocate the use of the synchronous model for programming interactive distributed multimedia applications and investigate whether it can guarantee the properties we are interested in. Synchronous languages were originally proposed for programming real-time reactive embedded systems. They rely on the *synchronous hypothesis* which states that programs take no time to produce

outputs when reacting to inputs. The precise notion of time in these languages is suitable for programming operations that should be performed respecting a given timing constraint, which are common case in multimedia.

In this thesis, we approach the problem of developing distributed applications in two steps. First, we tackle local applications and explore how synchronous languages can provide the support necessary for programming deterministic interactive multimedia applications. Second, we investigate how we can explore the advantages of the local case to the distributed setting while guaranteeing the consistency property. Here we have used the synchronous programming language Céu. However, our theoretical findings extend to any synchronous language that can provide properties that Céu's semantics guarantees [17].

One of these theoretical findings is the suitability of Céu for implementing some of the most common causal relationships among media objects. We discuss in this work how Céu constructs and semantics can be used for implementing the operations defined in the Interval Expression model [2], which is a model based on set of operators that expresses causal relations between intervals.

Céu-Media, a library for programming local multimedia applications in Céu, is the practical result of the first part of this thesis. It is an evidence that our argument in favor of synchronous languages holds: with Céu-Media, we managed to reproduce the accuracy and determinism of Céu's semantics in the final multimedia presentation output, thus guaranteeing determinism for local applications.

For the distributed scenario, we assume network architectures with no QoS guarantees. Several works approach such networks using the GALS (*Globally Asynchronous, Locally Synchronous*) architectural style [1]. In GALS systems, computations within individual synchronous nodes are deterministic, with the communication latency as the only source of non-determinism. Mars, the practical result of the second part of this thesis, is a middleware that follows the GALS style and supports consistent execution (following the timing-sequential consistency model) of distributed interactive multimedia applications. Moreover, the programming model promoted by Mars separates the concerns regarding application logic and inter-application communication bindings. The source code of a Mars application has no explicit communication primitive. In fact, these codes can be compiled and executed as a local application. The specification of how processes communicate is external to the application code.

In sum, in this thesis we advocate the use of the programming model

of synchronous languages and a consistency model that guarantees total ordering and timing synchronization of operations for programming distributed applications. We argue that by combining these two models, it is possible to guarantee deterministic and consistent executions without having to deal with low-level synchronization and communication issues. We discuss the gains and limitations of our proposal by investigating how some real-world use cases proposed by the research community can be developed using Céu-Media and Mars.

## 1.1 Outline

The rest of this thesis is organized as follows: Chapter 2 presents our theoretical background. Chapter 3 discusses the suitability of Céu for programming multimedia applications. Chapter 4 presents our approach to explore the synchronous programming model in the multimedia domain and presents Céu-Media. Chapter 5 describes how we have approached distributed settings and presents the Mars middleware. Chapter 6 describes some use cases defined by the research community and shows how they can be developed using Mars. Chapter 7 discusses related works and compares them to our proposal. Finally, Chapter 8 presents our final remarks and points out future works.

# 2
# Background

There are several types of multi-device applications. Levin [10] proposes the 3C framework as an attempt to categorize these applications in three groups. As pointed out by the author, these categories are not mutually exclusive, i.e., applications can lie in the intersection between them.

The first group, called *consistent*, refers to applications that replicate the same experience among different devices, adjusting the content to accommodate device-specific attributes. Spotify[1] is an example of consistent application, because it offers the same experience to users (discover and listen to music) in different devices, but adapts its interface according to the features of each appliance.

The second group is called *continuous* and allows users to transfer and continue the same activity between several devices. An e-commerce service which allows users to choose an item to buy in a given device and to complete the purchase in another is an example of continuous application.

The third group, called *complementary*, is our focus in this thesis. In complementary applications, each device complements one another creating an experience as a connected group. YouTube[2] and Netflix[3] are well-known examples of complementary applications: both allow users to control the playback of videos in a device (usually a smart TV) by using their smartphones.

Complementary applications are typically designed to run on multiple devices at the same time. The interactions among devices can either be collaborative-based, in which each device has its own role and works collaboratively to construct the whole user experience, as in Figure 2.1(a); or control-based, in which an application running on a device controls part of the exhibition on another device, as in Figure 2.1(b).

This Chapter reviews the theoretical background used throughout this research. As our proposal for supporting the programming of these applications relies on the use of synchronous languages, we first describe their main characteristics in Section 2.1. Then we present an overview of current approaches for programming multimedia applications in Section 2.2. Finally, we introduce

[1] https://www.spotify.com/
[2] https://www.youtube.com/
[3] https://www.netflix.com/

2.1(a): Car racing multiplayer game [18].

2.1(b): Remote control second screen application [19]

Figure 2.1: Example of complementary applications

the synchronous language Céu in Section 2.3. We focus our discussion on the problem of guaranteeing determinism (for local applications) and consistency (for distributed systems) as defined in Chapter 1.

## 2.1 Synchronous Languages

Synchronous reactive languages [20] (*synchronous languages*, as shorthand) rely on the *synchronous hypothesis* [12] which considers that programs produce outputs synchronously with their inputs. Reactive languages divide computations into a sequence of discrete steps called *reactions*. Each reaction executes to completion before the system can process any other input. The synchronous hypothesis adds the constraint that inside each reaction the time does not advance. In practice, this model assumes that computing reactions is much faster than the minimum time interval between external events, which is a feasible assumption in real-time embedded systems [21]. Esterel [22], Lustre [23], StateChart [24], Céu [25] are some examples of synchronous languages.

The guarantees provided by most synchronous languages can solve some problems of the multimedia programming field. In synchronous languages, time advances in a sequence of discrete input events, defining what is known as *logical time*—also known as logical control.

Timing is crucial for multimedia applications. Some authors regard to these applications as soft real-time systems [26, 27, 28]: the correctness of their executions depends not only on the accuracy of computations, but also on the time the result is presented [29]. Consider the rendering of a media file having a video and an audio streams. For a player correctly present this file, it should be able to decode both streams and render each video buffer and audio sample respecting their timestamps. The logical notion of time supplants the physical notion for programming such scenarios [30].

Most synchronous languages have deterministic semantics, which guarantees that multiple executions of programs always yield the same output. Likewise, it guarantees that the execution of the same program in different (but compliant) implementations of compilers or interpreters also yields the same output. Furthermore, it supports the implementation of validation tools that statically check programs for analysing their properties. In the embedded systems domain, these tools may be used for checking programs correctness. In the multimedia domain, they can be used for ensuring presentation properties (e.g., audio overlapping, video/images shadowing, contradictory constraints).

Synchronous languages have native support for concurrency, while preserving determinism. This means that one can develop programs that concurrently react to multiple events and still have deterministic behavior. Traditional multimedia languages such as NCL and SMIL allow the programming of concurrent lines of execution, but their predictability cannot be guaranteed (next section discusses some practical examples of this issue).

Support for event handling, in general, is a major concern of reactive languages. The programming of event-driven applications using traditional programming models is typically performed around the notion of asynchronous callbacks. One of the main issues when using callbacks is that program control jumps around multiple functions, leading to codes that are hard to follow and/or understand. In fact, control flow is driven by events and not by an order specified by the programmer. Synchronous languages overcome these problems by providing abstractions to express how programs should react to events. Besides, compilers usually guarantee safe access to shared variables, which has the advantage that programmers do not need to worry about the order and computation dependencies [31].

The approach of applying synchronous languages in the multimedia field is not novel. In the 90's, several authors explored the use of these languages for addressing the problem of real-time synchronization of streamed media content [31, 32, 33, 34, 35, 36]. ChucK [37], Pure Data [38], Csound [39], Faust [40] and SuperCollider [41] are some examples of synchronous DSLs developed for audio processing (also know as music programming languages). Because human hearing can detect even small latencies and delays in audio signals, the use of the synchronous approach represents an interesting alternative for providing timing guarantees over sample-level operations in the audio signal.

Smix [30] is a more recent proposal for high-level multimedia programming that also relies on the synchronous hypothesis. This DSL has been proposed as an alternative for traditional informal and ambiguous high-level multimedia languages and it was designed to have deterministic semantics since

its conception. However, it has no support for distributed applications.

These works help to illustrate how the research community has been for long investigating the use of synchronous languages for approaching problems of the multimedia field. However, none of them has explored the use of these languages in the context of programming interactive multi-device applications. They all have in common the assumption that the characteristics of synchronous languages constitute a suitable framework for programming the control part of multimedia systems. Here we borrow this assumption under the programming perspective and apply it in the distributed domain. However, approaching distributed systems using synchronous languages is not straightforward.

### Synchronous Programming and Distributed Systems

In the late 90's, the arising of Integrated Architectures led to the development of safety critical embedded systems composed of several nodes that communicate to perform a given function [42]. The deployment of synchronous applications in such architectures rises some issues that has driven several researches. Asynchronous interactions [1, 43], synchronous semantic preservation [44, 45], automatic distributed code generation [46, 47, 48], verification of synchronous distributed applications [49, 50] are some examples of these problems. Here we focus on the asynchronous interactions issue.

To better frame this discussion, let's consider the abstract service model composed of three layers depicted in Figure 2.2. It aims to better characterize the different models involved when developing synchronous distributed applications. Layer 0 comprises network architectures, that is, the set of protocols, connections, controllers, and guarantees that a given platform implements for allowing data to flow from a node to others. In Layer 1 lies the Distributed Synchronous Model of Computation (MoC) with two main goals. First, it implements a set of techniques for properly realizing the synchronous semantics in a distributed setting using the services provided by a particular network architecture. Second, it hides the complexities of the underlying network by offering a set of architecture-independent programming interfaces. Programmers develop applications (Layer 2) targeting a specific distributed synchronous MoC, which has the advantage of shielding softwares from the idiosyncrasies of a particular network architecture [51].

We can roughly divide network architectures into two main categories: those that can provide accurate clock synchronization and/or timing guarantees and those that cannot.

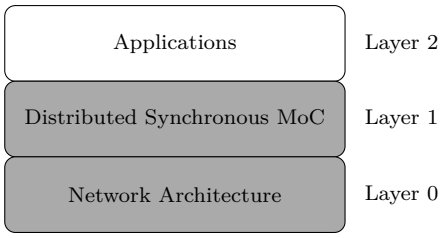


Figure 2.2: Three layers abstract service model for programming distributed synchronous applications.

Within the embedded software community, the TTA (Time-Triggered Architecture) [52] is one of the most remarkable example of the first category, followed by a more recent proposal called PALS (Physically Asynchronous Logically Synchronous) [53]. TTA systems assume a maximum network delay and rely on the notion of physical time consistently maintained synchronized throughout nodes. The PALS architecture is similar to TTA, but it makes stronger assumptions and has the abstraction of perfectly synchronized virtual clocks that drive computations in individual nodes. These timing guarantees support the deployment of systems in which all nodes operate in lockstep, changing their state synchronously [51].

Both architectures favor a distributed synchronous MoC whose semantics is close to those of synchronous languages targeting local applications. For instance, in [54] the authors discuss an approach for the design and implementation of distributed applications based on the synchronous language Lustre with a few extensions targeting the TTA architecture. These extensions aim to direct the compiler for correctly generating distributed code and *"do not change the high-level (logical-time) semantics of Lustre"* [54]. The works described in [55, 56, 57] are other examples of proposals that use TTA or PALS for supporting the development of distributed synchronous MoCs.

IntServ [58] and ATM [59] are other examples of networks that can provide some QoS (Quality of Service) guarantees (e.g., maximum network transmission delays) and could be used as target architectures for soft real-time applications (e.g., multimedia). How to exploit the characteristics and guarantees of these architectures to provide an appropriate distributed synchronous MoC is still an underinvestigated problem.

Architectures that can provide timing guarantees rely on very strong requirements on their underlying networks. Some of them are not feasible to assume in unmanageable environments such as the Internet or even in wireless local networks [51, 60]. Network architectures in the second category, that make no assumption regarding clock synchronization, clock paces or bounded communication delays (e.g., LTTA (Loosely Time-Triggered Architecture) [60],

Ethernet, WiFi, Internet) generally implement the GALS (Globally Asynchronous Locally Synchronous) [1] architectural style.

GALS is an alternative for developing systems in which individual modules take advantage of the synchronous approach and the communication latency and jitter are usually the only source of non-determinism. The GALS architectural design was originally proposed for programming multi-clock digital circuits, in which each synchronous block has its own clock running in its own frequency and they are interconnected through an asynchronous bus [1]. Figure 2.3 schematically illustrates the idea.

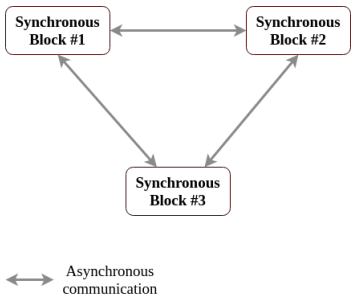


Figure 2.3: High-level diagram of a GALS multi-clock chip [1].

There are several flavors of Layer 1 systems that implement a GALS-like MoC. These systems may offer a programming interface based either on a synchronous or asynchronous framework. In the former approach, regular synchronous languages and tools are stressed for accommodating the asynchronous behaviour. Multiclock Esterel [61], CRP [62] and CRSM [63] are examples of Esterel-like languages that follow this concept. On the one hand, the asynchronous communication may be modelled using regular events without modifying the semantics of the languages, on the other hand, properties like liveness and fairness cannot be properly checked exactly because their semantics do not consider the distributed aspect of the system [64].

Alternatively, there are asynchronous languages designed with native support for programming GALS systems. Such languages have constructs and abstractions to reason about asynchronous concurrent systems. Furthermore, asynchronous verification frameworks can verify complex properties of these systems, for instance, succession of events in time, infinite executions and fairness [64]. However, these frameworks tend to be more complex when compared with their synchronous counterpart. SystemJ [65], DSystemJ [66] and GRL [64] are examples of asynchronous languages for GALS.

In the multimedia domain, network architectures that have timing guarantees could be used for devising a distributed synchronous MoC that provides some level of distributed playout synchronization. However, the global synchronization can introduce coordination issues. For instance, if all nodes are in the

exact same global state (logical time), concurrent users' interactions in a given state, but in different nodes, should be considered as occurring in parallel. This can lead to the classical distributed consensus problem[4] [67], with the peculiarity that the multimedia presentation can halt if the system takes too many rounds to reach the consensus.

Architectures without timing guarantees make the problem of implementing synchronized distributed playouts harder to solve, but they accommodate the implementation of techniques for enforcing total or partial ordering of events. In this work, we investigate how to devise a suitable synchronous MoC for programming interactive distributed multimedia applications in networks of this category. The Layer 1 GALS middleware Mars, which offers a programming interface based on the Céu language, is the result of this investigation.

## 2.2 Overview of Distributed Multimedia Programming Approaches

Traditionally, multimedia (local or distributed) applications are programmed either by using multimedia languages or by using general purpose languages supported by specialized frameworks. In this section we review these approaches and discuss how they fail to guarantee the determinism and consistency properties. Next chapter takes back the code excerpts discussed in this section and explains how the synchronous programming model solve their idiosyncrasies.

### 2.2.1 Multimedia Languages

Multimedia languages have been traditionally designed aiming to hide the complexity of implementing low-level operations and synchronization of media content by using high-level abstractions. Here we take NCL and SMIL as representative examples of multimedia languages. NCL is an ITU-T Recommendation [68] and adopts a synchronization model based on causal sentences: `link` elements define a set of conditions (e.g., `onBegin, onEnd, onPause`) that, when satisfied, triggers a set of actions (e.g., `start, stop, pause`). SMIL is a W3C Recommendation that has a constraint-based synchronization model: temporal containers (`par, seq` and `excl`) and attributes (e.g., `begin, end, dur`) define a set of constraints that must be satisfied at runtime.

Ambiguity, caused by the lack of deterministic semantics, is a problem of most high-level multimedia languages [30]. In general, these languages do not

[4] It is well-known that the distributed consensus problem is undecidable in the case of asynchronous communication with at least one faulty node. For the synchronus case, there are known solutions, but they usually requires some rounds to reach the consensus.

```
1  <ncl>
2    < ... >
3    <body>
4      <port id="p1" component="m1"/>
5      <port id="p2" component="m2"/>
6      <port id="p3" component="m3"/>
7      <...>
8      <port id="pN" component="mN"/>
9
10     <media id="m1" .../>
11     <media id="m2" .../>
12     <media id="m3" .../>
13     <...>
14     <media id="mN" .../>
15   </body>
16 </ncl>
```

Listing 2.1: An NCL code that starts $N$ media objects when the program begins. The language does not guarantees that all objects start at the same time.

have a well-defined execution model (evidenced by the number of corner cases in their manuals) hindering formal definitions of their semantics. Therefore, their specifications lie in verbose manuals written in a natural language using normative definitions. In some cases, the ambiguity is acknowledged by the specification itself: the SMIL 3.0 manual explicitly states that some constructs of the language may have different interpretations in different players [14].

Besides ambiguity, the absence of a precise definition of the execution model can also lead to synchronization problems. Let's take as example the passage of time in NCL. Following only its semantics (that is, without relying on a specific player) there is no way to program a set of objects to start exactly at the same time. The problem is that the language semantics does not enforce that the presentation time should not advance while players react to an event. Even though occasionally two objects indeed start at the same time, the language does not impose that they should be rendered at the same pace. Therefore, one cannot assume that they will remain in sync.

To make matters concrete, let's consider the NCL code depicted in Listing 2.1 (the following discussion has been adapted from [69]). This code excerpt specifies an NCL program that has $N$ media objects (`m1, m2, m3, ..., mN`)—lines 10–14. As soon as it starts, all $N$ objects should be started (note the `port` statements in lines 4–8) intuitively at instant $0s$. Because the language does not impose, the presentation time may keep running while a player starts each media object. Thus, even though for a small $N$ it is possible that all objects start at the same time, as $N$ gets larger, the dyssynchrony problem becomes more evident. That is, delays accumulate and some objects that are executed later will have a start time different from those started earlier.

Now let's examine another problem also caused by the lack of a precise execution model in NCL: the order of evaluation and execution of `link` elements is arbitrary. Bearing this in mind, consider the code in Listing 2.2. It defines a simple program that has only one media object (`m1`, line 5) that is started when the program starts (`port` statement in line 4). The `link l1` in lines 6–9 defines the following causal relationship: when object `m1` begins, stop `m1`; and the `link l2` in lines 10–13 defines another relationship: when object `m1` begins, start `m1`.

```
1  <ncl>
2    < ... >
3    <body>
4      <port id="p1" component="m1"/>
5      <media id="m1" .../>
6      <link id="l1" xconnector="onBeginStop">
7        <bind role="onBegin" component="m1"/>
8        <bind role="stop" component="m1"/>
9      </link>
10     <link id="l2" xconnector="onBeginStart">
11       <bind role="onBegin" component="m1"/>
12       <bind role="start" component="m1"/>
13     </link>
14   </body>
15 </ncl>
```

Listing 2.2: NCL code with multiple accepted behaviors. The player can execute `l1` and `l2` in arbitrary orders when `m1` starts at the beginning of the program. Depending on the player's choice, the final result may be different.

There are different accepted executions for this simple program. When `m1` starts, the triggering conditions of `l1` and `l2` are satisfied. At this point, that are two scenarios: the player executes first either `l1` or `l2`. The simpler case is when it executes `l2` first. In this situation, the `start` in line 12 is ignored (because `m1` is already playing). The player then executes `l1` and stops `m1` due to the `stop` action in line 8.

The second case is more tricky. If the player executes `l1` first, the behavior depends on whether it implements state changes synchronously or asynchronously. In the synchronous case, the `stop` in line 8 immediately stops `m1` and, when `l2` is executed, the `start` in line 12 immediately starts it, causing the recursive triggering of `l1` and `l2`. In this situation, if `l1` is *always* executed before `l2`, we have an infinite loop. Now let's consider the asynchronous case in which the `stop` in line 8 schedules the stop of `m1`. The behavior when the player evaluates `l2` and is about to execute the `start` in line 12 depends on whether the asynchronous `stop` has completed or not. If the `stop` completes before the

`start`, then `m1` is started again, triggering `l1` and `l2` recursively. Otherwise, the `start` should be ignored because `m1` is still playing.

The stricter SMIL's synchronization model reduces, but does not eliminate, problems similar to those discussed above. The language has constructs to specify that multiple objects should have exactly the same start time and also has attributes for specifying the maximum tolerable dyssynchrony among objects. However, its synchronization model allows the programming of inconsistencies that may be difficult to detect statically [70, 71, 72]. Furthermore, SMIL also admits an implementation-dependent propagation delay between the generation of an event and its processing, which may lead to different behaviors in different players [14].

As an example, consider the SMIL code depicted in Listing 2.3 (we have adapted this example from the official test suite [14]). The **`par`** element in lines 3–6 defines a temporal container that may playback its children in parallel. The object `img1` (line 4) starts $3s$ after the beginning of its parent composition (**`begin="3s"`**) and ends immediately, because its **`end`** attribute is also set to $3s$. The object `img2` (line 5) is set to start when the `img1` starts (**`begin="img1.beginEvent"`**) and it has $4s$ duration (**`dur="4s"`**).

```
1 <smil>
2   <body>
3     <par id="par1">
4       <img id="img1" begin="3s" end="3s" ... />
5       <img id="img2" begin="img1.beginEvent" dur="4s" ... />
6     </par>
7   </body>
8 </smil>
```

Listing 2.3: SMIL code with multiple accepted behaviors. The object `img2` may be presented or not depending on how the player processes internal timing events.

The SMIL specification defines that a temporal container should end if the following two conditions hold:

i. it has no children being executed;

ii. no children has its **`begin`** time resolved.

According to [14], *"the delivery of the [img1].beginEvent to the [img2] element may not occur until after the par has ended at 3s"*. This may happen if between the raising of the event **`img1.beginEvent`** and its processing by `img2`, the player checks conditions $i$ and $ii$: $i$ is *true* because `img1` has $0s$ duration and `img2` has not yet started; $ii$ is *true* because the restriction **`begin="img1.beginEvent"`**

is said *unresolved* until the given event is processed. Therefore, a compliant player must show `img1` for $0s$ and then either end the container or show `img2` for $4s$.

There are some proposals for solving the ambiguities discussed in the examples above. For the NCL case, in [69] the authors propose converting NCL documents to the synchronous language SMIX, aiming to fix semantic problems. For the SMIL case, the specification itself suggests that *"it is desirable for [players] to behave as if they responded to the internal timing events instantaneously[...]"*. That is, in both cases, the proposed solutions resort to the implementation of the synchronous execution model advocated in this thesis[5].

Following the discussion in [30], multimedia languages are over-engineered, which in part justifies their complexity and ambiguity. That is, *"their specifications try to accommodate many, sometimes conflicting, [features]"* [30]. Sometimes (notably SMIL, NCL, and HTML), the standardization process itself, led by heterogeneous groups, leads to ambiguous specifications: the documents that define the languages must somehow embrace, into coherent definitions, interests of all people involved in the process. The lack of formal semantics makes difficult the detection of conflicting definitions during this process.

Several works in the literature address these problems by proposing an alternative formal and deterministic semantics [74, 75, 76, 77] to support the implementation of tools that statically check presentation properties (audio overlapping, video/images shadowing, contradictory constraints) [78, 79, 70, 80]. Due to the complexity of the languages, these works tend to consider just a subset of their constructs. Another drawback is that there is no guarantee that the properties checked by these tools will hold at runtime, because players tend to implement the official (and ambiguous) semantics.

Now let's consider the support for programming interactive distributed applications. SMIL, HTML, SVG, X3D and XMT have no native support for multi-device applications. On the contrary, NCL and IPML (a SMIL-based language) natively implement declarative constructs for distributed applications: both languages allow one to program in which device a given media content should be presented. While NCL admits the programming of interactive distributed applications, it is not clear whether IPML supports distributed interactivity (the authors state nothing about interactivity in their proposal). Nevertheless, neither NCL nor IPML enforce total ordering of events

[5]On the contrary of all other multimedia languages cited in this section, the SVG specification defines that when reacting to events, all consequential actions must be performed at the same timing instant [73], assuming implicitly the synchronous model.

in distributed applications. So, even if peers running applications developed in these languages behaved deterministically (which is not the case) the system would fail to guarantee consistent executions according to the consistency model defined in [4] and used in this thesis. That is, it is possible to develop distributed applications in NCL and IPML in which each peers' view of the order that events have occurred may be different from others.

Finally, most systems that adopt one of these multimedia languages also admit the use of a scripting language to overcome the limited-expressiveness problem of DSLs. In some cases, there are APIs specifically designed aiming to support the programming of distributed multimedia applications. Consider the case of HTML and JavaScript. At the time of this writing, there are at least two W3C Working Groups whose main focus is to propose JavaScript APIs for supporting multi-device presentations in the Web ecosystem: the Multi-Device Timing Community Group[6] and the Second Screen Community Group[7]. The first group is proposing the *TimingObject* concept which is capable of proving a synchronized clock among different devices. The second group's proposal consists of the *Presentation* API, which aims to favor the development of applications that present web content on a secondary display connected to a device.

Each of these proposals target a specific feature and does not aim to implement a full-fledged framework for interactive multi-device applications. The *TimingObject* supports the development of distributed multimedia presentation that can provide some level of synchronized playouts in different devices. The *Presentation* API allows a device to control the presentation in another device. Functionalities such as ordering of messages should be implemented from scratch, similar to the approach discussed in next section.

### 2.2.2 General Purpose Languages

Programming multimedia using general purpose languages usually implies in using specialized frameworks. GStreamer[8], FFmpeg[9], libav[10], libVLC[11], DirectShow[12] and AV Foundation[13] are some examples of multimedia frame-

[6] `https://www.w3.org/community/webtiming/`
[7] `https://www.w3.org/community/webscreens//`
[8] `https://gstreamer.freedesktop.org/`
[9] `https://ffmpeg.org/`
[10] `https://libav.org/`
[11] `http://www.videolan.org/vlc/libvlc.html`
[12] `https://msdn.microsoft.com/en-us/library/windows/desktop/dd375454(v=vs.85).aspx`
[13] `https://developer.apple.com/av-foundation/`

works for general purpose languages. Flexibility is a strength of this approach, but it comes with the price of complexity.

Most of these frameworks have an API that exposes low-level operations to programmers. For instance, functions for manipulating video buffers, audio samples, color spaces, bitrates are typically implemented by all of them. In general, they favor operations at intra-stream level over the composition of multiple objects. As a consequence, high-level operations, such as synchronization of different streams, users' interaction, detection of the end of objects (considering a single object may have multiple streams) should be programmed on top of the low-level API.

In applications that handle concurrent multiple media objects, the use of threads is common. In these cases, it is possible to guarantee the determinism, but it should be done programmatically by synchronizing threads and access to shared variables, which can be complex [16].

These frameworks have limited support for distributed applications, focusing mainly on media streaming. GStreamer goes a step further by implementing clock synchronization in different devices for synchronized rendering. However, there is no native support for communication among devices that guarantees ordering of messages (consistency). Again, programmers can enforce consistency by implementing a communication layer that ensures this property or using a communication library that guarantees it. Note that this reinforces our point of lacking unified proposals for programming distributed multimedia applications.

## 2.3 Céu

Céu [25] is a reactive synchronous language originally developed as a safe alternative for programming soft real-time embedded systems. Céu programs advance in a sequence of discrete reactions to external events received from their environments. Reactions are synchronous (i.e., instantaneous, according to the synchronous hypothesis), run atomically and to completion.

Céu has been designed for control-intensive applications, therefore it has structured mechanisms, such as `await` (to suspend lines of execution) and `par` (to create logical concurrent lines of execution) which favor one to write code in direct style, as opposed to the inversion of control in event-driven executions relying on callbacks. Let's take the example in Listing 2.4 to present the language. This Céu code generates a program that toggles Led1 and Led2 on and off at each $2s$ and $4s$, respectively, until a key is pressed.

This code first declares the `input` event `KEY` (line 1). In lines 2–19 there

```
1  input void KEY;
2  par/or
3  do                  /* trail 1 */
4    loop do
5      await 2s;
6      _Led1_on()
7      await 2s;
8      _Led1_off()
9    end
10 with                /* trail 2 */
11   loop do
12     await 4s;
13     _Led2_on()
14     await 4s;
15     _Led2_off()
16   end
17 with                /* trail 3 */
18   await KEY;
19 end
```

Listing 2.4: A CÉU code that toggles Led1 at each $2s$ and Led2 at each $4s$ until one presses a key.

is a parallel composition, that creates concurrent lines of executions known as *trails*. In this example, we use a `par/or` (*parallel-or*) composition, which finishes whenever one of its trails finishes. CÉU also has the compositions `par/and` (finishes only when all of its trails finish) and `par` (never finishes).

The `par/or` composition in Listing 2.4 creates 3 trails. The first one (lines 3–10) has a `loop` (lines 4–9) that suspends its execution for $2s$ (line 5), toggles Led1 on (line 6), suspends its execution for another $2s$ (line 7) and toggles Led1 off (line 8), restarting the loop. The second trail (lines 10–17) is similar to the first, but it toggles Led2 on and off at each $4s$ (lines 12 and 14). Note that neither the first nor the second trails finish because both run an infinite loop. Finally, the last trail (lines 17–19) simply suspends its execution until the program receives a `KEY` event (line 18), which causes its end and, consequently, finishes the whole composition (i.e., other trails are aborted). At this point, all trails rejoin at line 19 and the program finishes.

### Execution Model

CÉU's semantics enforces deterministic execution of programs even when using parallel compositions: when multiple trails are activated by the same event, they are scheduled in lexical order (the order they appear in the source code). CÉU also detects at compile time conflicting access to shared variables to guarantee safe executions [81]. However, this check is optional and is not essential for providing deterministic behaviors. The following algorithm

summarizes how a CÉU program executes:

1. The program initiates a *boot reaction* in a single trail. Parallel compositions may create new trails.
2. Active trails execute until they block in an `await` or terminate.
3. When all trails block (i.e., the reaction finishes) the control goes back to the environment.
4. If an event $E$ occurs, all trails waiting for $E$ are resumed in order and the execution goes back to the step 2.

Note that following this execution model, reactions are triggered always in response to a single external event. Suppose that a CÉU program receives the event $E$. Thus, according to step 4, all trails waiting for that event wake and execute. If during this reaction the program receives an event $E'$, then it is queued and processed in the next reaction, i.e., trails waiting for $E'$ will wake when the current reaction ends.

Given this execution model, combined with the synchronous hypothesis, Figure 2.4 depicts a timeline that precisely represents the operations (and their order) executed by the program in Listing 2.4. Here it is worth noting that, as other synchronous languages, CÉU adopts a logical notion of time. Thus, for instance, the $2s$ written in lines 5 and 7 of the Listing 2.4 corresponds to two logical seconds (i.e., two occurrences of the event *second*) and not necessarily two physical seconds. In CÉU, `await` statements are the only instructions that actually takes time and all other statements are instantaneous (that is, the logical time does not advance while the program executes them).

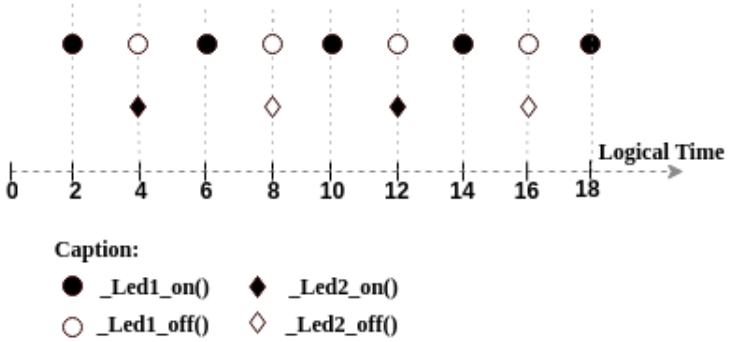


Figure 2.4: Timeline of the toggling Leds program.

So, the program in Listing 2.4 executes as follows. At exactly $2s$ of logical time the program executes the function `_Led1_on()`. At $4s$, the program executes two functions in the following order: it first calls `_Led1_off()` and

then `_Led2_on()`. At $6s$, the program calls `_Led1_on()`. Again, at $8s$, it executes two functions (in this order): first `_Led1_off()` and then `_Led2_off()`. The deterministic semantics of CÉU guarantees that this execution pattern repeats indefinitely until one presses a key.

### Abortion and Finalization

CÉU **`par/or`** construct provides a natural means for implementing an orthogonal abortion mechanism. Consider again Listing 2.4, which finishes when one presses a key. Let's adapt that program to also finish when one presses a mouse button (`MOUSE_CLICK` event). In this situation, it is enough to add a fourth trail that just waits for the event `MOUSE_CLICK` and finishes, without having to tweak any other trail (e.g., add synchronization primitives or state variables). Orthogonal abortion is an intrinsic characteristic of synchronous languages and cannot be expressed effectively when using threads [82].

However, preempting the execution of a trail may lead to undesirable situations. A classical example is when a trail that has opened a file is aborted without having the chance to properly close it. For such cases, CÉU has the **`finalize`** construct that always executes when its enclosing block ends. Listing 2.5 illustrates an example. The first trail of the **`par/or`** composition opens a file using the C `fopen` function, which returns a pointer to a file descriptor (line 5). When the first trail ends, either by receiving the $E$ event (line 10) or it is aborted after $10s$ (line 12), the **`with`** clause of the **`finalize`** construct is executed, ensuring the program always closes that file (lines 4–8).

```
1  <...>
2  par/or do
3    var _FILE *f;
4    finalize
5      f = _fopen (<...>);
6    with
7      _fclose (f);
8    end
9    _fwrite (<...>, f);
10   await E;
11 with
12   await 10s;
13 end
```

Listing 2.5: The finalize block always executes whenever the first trail ends.

### Internal Events

In addition to external events, CÉU implements the concept of internal events that are emitted internally by the program via **emit** statements. CÉU runtime processes internal events in a stack-based manner, instead of the queue-based processing of external events. Internal events serve as a signalling and communication mechanism among trails and produce micro-reactions within external reactions. They can be used to implement a limited form of subroutines, as depicted in Listing 2.6.

```
1  event (int *) inc;
2  par/or do
3    var int* p;
4    every p in inc do
5      *p = *p + 1;
6    end
7  with
8    var int v = 1;
9    emit inc (&v); //v == 2
10   emit inc (&v); //v == 3
11 end
```

Listing 2.6: CÉU processes internal events in a stack-based manner, which can be used for implementing "subroutines".

Line 1 declares the internal event `inc`. The CÉU **every** statement continuously wait for its identifying event, executing its body on each occurrence (this statement is known as *event iterator*). The first trail of the **par/or** composition uses an event iterator to react to occurrences of the `inc` event and increments the value received as reference (lines 4–6). In practice, this trail behaves as if it had defined a subroutine called `inc`.

The second trail defines the variable `v` with the value 1 (line 8) and emits twice the internal event `inc`, passing `v` by reference (lines 9–10). Whenever a trail emits an internal event, it pauses and the control goes to any previously executed trail which is waiting for that event. In the example, when the second trail emits the `inc` event in line 9, it pauses, the **every** iterator in line 4 wakes, executing its body, and only after it finishes the second trail is resumed. At this point, the variable `v` has the value 2. When the second trail emits that same internal event in line 10, it pauses and the **every** block executes again, which makes `v` to hold the value 3.

CÉU supports nested emitting of internal events. Thus, the event iterator in the first trail could emit another internal event, which would create a new level in the stack. The stack serves as a record for nested micro reactions.

A limitation of this form of subroutines is that it cannot express recursion, because an `emit` to itself is always ignored (a running trail cannot be waiting on itself).

### Compilation

The compilation of a CÉU source code is performed in two stages. First the CÉU compiler generates a corresponding C program, that is then compiled, using a standard C compiler, to machine code. During the first phase, CÉU checks the code to make sure that the properties guaranteed by its semantics (synchronicity, termination, consistency and determinism) indeed hold, otherwise it rejects that code and the compilation fails. Here, exceptions are native C calls (any statement starting with underscore)—CÉU passes those statements *as is* to the C compiler. If on the one hand this integrates seamlessly with C favoring the calling of native functions, on the other hand these calls cannot be checked by CÉU. Thus, if a native function performs blocking operations or takes a non-negligible time to execute, the logical time may diverge from the physical time. But from the CÉU perspective, all those calls are considered instantaneous.

Next chapter discusses the suitability of CÉU for programming some of the most used patterns in multimedia applications.

# 3
# Céu and Multimedia

In this chapter we discuss the advantages of using Céu and its synchronous semantics in the multimedia domain. First, in Section 3.1 we demonstrate how one can express common causal relationships among media objects in Céu. Second, in Section 3.1 we discuss the implementation of a minimalist media player controller in Céu. And third, in Section 3.3 we discuss through examples how the synchronous semantics of the language prevents the problems discussed in Section 2.2.1.

## 3.1
## Expressing Causal Relationships in Céu

Programming multimedia applications is all about defining how media objects relate to each other. In a seminal work, Allen introduced an interval-based temporal algebra that defines a set of thirteen relations that "*can be used to express any relationship that can hold between two intervals*" [83]. Several works use this algebra either as basis for a formal timing model [84, 85, 86] or as an evaluation criteria for the expressiveness of languages or frameworks [87, 88, 89].

However, using Allen relations for programming the behavior of a multimedia application faces several problems: "*First, the relations are descriptive: they do not reflect causal dependency between intervals, but they rather represent temporal coincidence. Second, since the relations depend on interval duration, changing duration may modify the relation that exists between the intervals. Third, composition based on the relations may lead to temporal inconsistency, because contradictory relations may be specified for intervals*" [2].

Here we use an alternative model proposed by Duda and Keramane, known as Interval Expression [2], as basis for our discussion regarding the suitability of Céu for programming multimedia applications. Similarly to Allen's model, the Interval Expression is also interval-based, but it uses causal relationships (instead of restrictions) for expressing multimedia presentations. Figure 3.1 depicts the operators defined in that model. Each operator takes time intervals as arguments and yield another interval as a result.

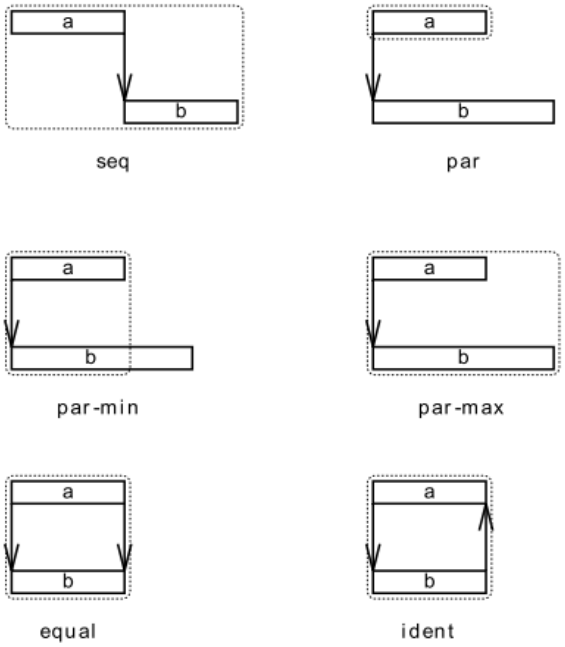


Figure 3.1: Temporal composition operators defined by Duda and Keramane [2].

The semantics of these operators is defined as follows:

- a *seq* b: the end of interval *a starts* interval *b*;
- a *par* b: the beginning of interval *a starts* interval *b*;
- a $par-min$ b: the beginning of interval *a starts* interval *b* and the result interval is stopped when the first of the two interval terminates;
- a $par-max$ b: the beginning of interval *a starts* interval *b* and the result interval is stopped when the last of the two interval terminates;
- a *equal* b: interval *a starts* and *stops* interval *b*;
- a *ident* b: the beginning of interval *a starts b* and the end of interval *b stops a*;

Table 3.1 depicts Céu patterns that implement each of these operators. For that discussion, suppose that Céu statements `a` and `b` are expressions that halt their trails to execute a media object and wake when the respective object finishes. And the `await FOREVER;` expression halts its trail indefinitely, i.e., it never wakes.

The following discussion briefly outlines the execution of codes in Table 3.1:

- **a *seq* b:** because there is no `await` between `a` and `b`, when the object `a` finishes, the object `b` starts immediately;

Table 3.1: Expressing the operators of Interval Expression in Céu.

| Relation | Céu | Relation | Céu |
|---|---|---|---|
| a *seq* b | `a;`<br>`b;` | a *par* b | `par do`<br>`  a;`<br>`with`<br>`  b;`<br>`end` |
| a $par - min$ b | `par/or do`<br>`  a;`<br>`with`<br>`  b;`<br>`end` | a $par - max$ b | `par/and do`<br>`  a;`<br>`with`<br>`  b;`<br>`end` |
| a *equal* b | `par/or do`<br>`  a;`<br>`with`<br>`  b;`<br>`  await FOREVER;`<br>`end` | a *ident* b | `par/or do`<br>`  a;`<br>`  await FOREVER;`<br>`with`<br>`  b;`<br>`end` |

- **a** *par* **b:** the `par` composition starts `a` and `b` together.
- **a** $par-min$ **b:** the `par/or` composition starts `a` and `b` together and finishes when the first of the two objects terminates.
- **a** $par - max$ **b:** the `par/and` composition starts `a` and `b` together and finishes when the last of the two objects terminates.
- **a** *equal* **b:** the `par/or` composition starts `a` and `b` together. The `await FOREVER` prevents the second trail to finish, therefore the composition ends when `a` terminates.
- **a** *ident* **b:** the `par/or` composition starts `a` and `b` together. The `await FOREVER` prevents the first trail to finish, therefore the composition ends when `b` terminates.

It worth mentioning that the semantics of the SMIL containers `<par>` and `<seq>` can be directly expressed by the Interval Expression operators $par - max/par - min$ (depending on whether the `endsync` attribute has the value `last` or `first`) and *seq*, respectively. Likewise, NCL connectors that have as condition `onBegin` or `onEnd` and as action `start` or `stop`[1] can be expressed by means of operators *seq*, *par*, *equal* and *ident*.

[1] By design, the Interval Expression model only relates the beginning and ending of intervals, therefore only NCL connectors whose conditions are `onBegin` or `onEnd` and actions are `start` or `stop` can be expressed by them. For instance, the operators *par* and *seq* correspond to the connectors `onBeginStart` and `onEndStart`, respectively. Mapping other NCL relationships is a matter of creating other operations with the corresponding semantics.

This discussion indicates that one can program in CÉU multimedia applications that causally relates media objects, similarly to NCL and SMIL. Section 3.3 supports this claim by discussing that the CÉU semantics avoids several ambiguities of these multimedia languages.

## 3.2 A Simplistic Multimedia Player in Céu

To demonstrate the use of CÉU constructs for programming multimedia, Listing 3.1 depicts the implementation of a simplistic player in CÉU used throughout this chapter. Assume that the CÉU expression `await Play (<URI>)` calls the function `Play` and halts the execution of its trail until the function returns. That function executes the media object pointed by `<URI>` and finishes when its presentation ends.

The code of this player interacts with other trails via four internal events: `start` (line 3), `stop` (line 5), `started` (line 10) and `stopped` (line 13). Trails emit the `start` event to signal to the player that it should start the execution of a given media object. The player stops the execution of that object when it receives the event `stop`. When the player starts to execute an object, it emits the event `started`. Finally, when the object stops (either naturally or due to a `stop` event) it emits the event `stopped`. Let's say that when the player is waiting for the event `start` it is in *sleeping mode*, and when it is executing an object it is in *playing mode*.

```
1  /* player.ceu */
2  loop do
3    var [] byte uri = await start;
4    par/or do
5      await stop;
6    with
7      par/and do
8        await Play (uri);
9      with
10       emit started;
11     end
12   end
13   emit stopped;
14 end
```

Listing 3.1: A CÉU player that starts an object when one emits the event `start` and finishes it when one emits the event `stop`. It also emits the event `started` when the object begins and the event `stopped` when it ends.

To implement this behavior, the player runs a loop which waits for the event `start` (line 3). When it wakes from this `await`, it creates a `par/or`

composition (lines 4–12) with two trails. The first trail simply waits for the event `stop` (line 5) to abort the presentation of the media object (i.e., this code uses the CÉU orthogonal abortion mechanism). The second trail, in parallel, calls the function `Play` (line 8) and emits the event `started` (line 10). Note that emitting a `start` event when the player is in *playing mode* has no effect, because it is no longer halted in the `await` in line 3. When the `par/or` composition ends, either because it has received the event `stop` (line 5) or because the media object has finished (line 8), the player emits the event `stopped` (line 13) and the loop completes an iteration, going back to the *sleeping mode*.

Consider a generic hardware to play multimedia files with two buttons: start and stop. Listing 3.2 implements a minimalist controller to this hardware using the player depicted in Listing 3.1.

```
1  par do
2    #include "player.ceu"
3  with
4    loop do
5      await START_BUTTON;
6      var [] byte file = <URI to play>;
7      emit start (file);
8      par/or do
9        await STOP_BUTTON;
10       emit stop;
11     with
12       await stopped;
13     end
14   end
15 end
```

Listing 3.2: A minimalist media player controller in CÉU.

The controller has a parallel composition with two trails. The first includes the code of the player (line 2). The second trail (lines 3—15) implements the controller logic. It runs a loop that waits the event `START_BUTTON` (line 5) to then emit the event `start` (line 7) leading the player to the *playing mode*. The trail then creates a `par/or` composition with two trails. The first waits the event `STOP_BUTTON` (line 9) to then emit the event `stop` (line 10) stopping the execution of the file. The second trail simply waits the event `stopped` to then finish. That is, this composition finishes either if one presses the stop button or if the media object finishes naturally. The loop completes an iteration and the controller is waiting again for the event `START_BUTTON`.

## 3.3 Fixing Semantic Problems of Multimedia Applications Using Céu

In this section, we take back the source codes discussed in Section 2.2.1 and present, side by side, alternative CÉU versions for them. Here our goal is to demonstrate that CÉU semantics shields programmers from the problems that arise when using non-deterministic multimedia languages. We focus mainly on the synchronous semantics rather than on the multimedia aspect of these codes. In fact, this discussion was the basis for our requirement analysis for developing a synchronous multimedia library able to guarantee deterministic behaviors for local applications.

First, let's consider again the NCL code in Listing 2.1, on page 27, which starts $N$ media objects when the program begins. Remember that the NCL semantics does not guarantee that all objects start exactly at the same time. Here we discuss how the synchronous semantics and the logical time guarantees the precise synchronization of the start time of multiple objects. Consider the code in Listing 3.3. It has a `par/and` composition in lines 1–10 which creates $N$ trails, each for playing a different media content. The program ends after the presentation of all objects.

```
1  par/and do
2    await Play (<URI_1>);
3  with
4    await Play (<URI_2>);
5  with
6    await Play (<URI_3>);
7    <...>
8  with
9    await Play (<URI_N>);
10 end
11
```

```
1  <ncl>
2    < ... >
3    <body>
4      <port id="p1" component="m1"/>
5      <port id="p2" component="m2"/>
6      <port id="p3" component="m3"/>
7      <...>
8      <port id="pN" component="mN"/>
9
10     <media id="m1" .../>
11     <media id="m2" .../>
12     <media id="m3" .../>
13     <...>
14     <media id="mN" .../>
15   </body>
16 </ncl>
17
```

Listing 3.3: A CÉU code that starts $N$ media objects when the program begins.

Let's analyze how this program guarantees the synchronization of the beginning of all objects according to the execution model discussed in Section 2.3. At logical instant $0s$, the program initiates the *boot reaction*, evaluates the `par/and` composition, and then starts to create the $N$ trails following the lexical order. First, it creates the first trail, which calls the `Play` function to start the object pointed by `<URI_1>`, and then halts until that object finishes (line 2). The program then creates the second trail, which also calls the

`Play` function, but it starts the object pointed by `<URI_2>`, and then halts waiting its end (line 4). The *boot reaction* follows, creating the third trail, executing it until it halts (line 6), to then create the next trail. This process goes on until the program creates and halts the last trail (line 9). At this point, the *boot reaction* finishes and the control goes back to the environment.

Note that the creation of all $N$ trails occurs during the *boot reaction*, that is, all calls the program does to the function `Play` happen at the same logical instant $0s$. This means that, from Céu perspective, all objects have exactly the same start time ($0s$), regardless of how large $N$ is[2].

Consider now the NCL code discussed in Listing 2.2, on page 28, which has different results depending on the order the player executes `link` elements. Because Céu has a deterministic trail scheduler, given an input event we can always accurately compute the sequence of operations the program executes. The code in Listing 3.4 is a Céu alternative for that NCL program.

```
1  par/and
2  do      /* link id=l2 */
3    every started do
4      emit start (<URI>);
5    end
6  with    /* link id=l1 */
7    every started do
8      emit stop;
9    end
10 with
11   #include "player.ceu";
12 with    /* port id=p1 */
13   emit start (<URI>);
14 end
15
```

```
1  <body>
2    <port id="p1" component="m1"/>
3    <media id="m1" ... />
4    <link id="l1"
5          xconnector="onBeginStop">
6      <bind role="onBegin"
7                    component="m1"/>
8      <bind role="stop"
9                    component="m1"/>
10   </link>
11   <link id="l2"
12         xconnector="onBeginStart">
13     <bind role="onBegin"
14                   component="m1"/>
15     <bind role="start"
16                   component="m1"/>
17   </link>
18 </body>
19
```

Listing 3.4: In Céu, the order of execution of trails is always known and the output can be previously computed.

The first trail implements the behavior of the NCL `link l2`: it reacts to each occurrence of the event `started` (equivalent to the NCL `onBegin` event), emitting the event `start` (lines 3–5). The second trail (lines 7–9) is similar to the first, but it emits the event `stop` (as the NCL `link l1`). The third trail imports the player in Listing 3.1. And the last trail simply emits the event `start` (line 13) which has the same effect of the NCL `port p1` statement.

[2] The pattern presented in Listing 3.3 behaves similarly to the SMIL `par` container with `endsync="last"`, regarding the beginning of its child elements.

During the *boot reaction* the program evaluates the `par/and` and creates the four trails, always following the lexical order. When the fourth trail emits the event `start`, the player which was in *sleeping mode* wakes and goes to *playing mode*, emitting the event `started`. The first and second trails, which are waiting for that event, wake in order. The first trail starts to react and emits the event `start`, but because there is no trail waiting for it (remember that the player is in *playing mode*) nothing happens. The second trail then wakes and emits the event `stop`. The player wakes again, but now it aborts the execution of the media object, and completes a loop iteration, going back to *sleeping mode*. At this point, all internal micro-reactions have finished, as well as the *boot reaction*. So, at the end of the *boot* the player is in *sleeping mode*.

Now let's make a slight change in the code of Listing 3.4: swap the first and second trails (Listing 3.5). This modification leads to a program with different execution.

```
par/and
do      /* link id=l1 */
  every started do
    emit stop;
  end
with    /* link id=l2 */
  every started do
    emit start (<URI>);
  end
with
  #include "player.ceu";
with    /* port id=p1 */
  emit start (<URI>);
end

```

```
<body>
  <port id="p1" component="m1"/>
  <media id="m1" ... />
  <link id="l1"
          xconnector="onBeginStop">
    <bind role="onBegin"
                  component="m1"/>
    <bind role="stop"
                  component="m1"/>
  </link>
  <link id="l2"
          xconnector="onBeginStart">
    <bind role="onBegin"
                  component="m1"/>
    <bind role="start"
                  component="m1"/>
  </link>
</body>

```

Listing 3.5: Swapping the first two trails leads to a different sequence of operations.

As in the previous example, when the fourth trail emits the event `start`, the player wakes and goes to *playing mode*, emitting the event `started` and waking the first two trails in order. The first trail wakes and emits the event `stop`. The player wakes due to that event, aborts the execution of the media object, and goes to *sleeping mode*. The second trail then wakes and emits the event `start`, which wakes again the player. At this point, the player goes to *playing mode* and emits the event `started` for the second time. This event wakes again the first trail, which emits the event `stop` and the player goes to *sleeping mode*.

At first sight, it seems this program goes into a loop when the second trail wakes again due to the second event `started`. But, as stated in Section 2.3, the use of internal events has an important limitation: recursion is not supported. Thus, the second trail cannot wake in this micro-reaction because it has been indirectly triggered by this very same trail (remember that a "*running trail cannot be waiting on itself*"). At this point all micro-reactions finish. At the end of the *boot reaction* the player is in *sleeping mode*, as in the previous example.

These two examples serve mainly to illustrate two points of the Céu semantics. First, changes in the order of trails can lead to different (but deterministic) executions. Second, the program never goes into a loop—as proved by Santos et. al, reactions in Céu always finish, that is, they never go into an infinite loop [17].

Now let's move to the SMIL code depicted in Listing 2.3, on page 29, in which depending on how the player processes internal timing events, the object `img2` may be presented or not. Listing 3.6 illustrates a version for that program written in Céu, whose well-defined execution model disallows that non-deterministic behavior. The outermost **`par/and`** implements the behavior of the SMIL `par1` composition: it executes in parallel the object `img1` and `img2`, the former according to restrictions **`begin="3s"`** and **`end="3s"`** specified by the SMIL element `img1`, and the latter according to restrictions **`dur="4"`** and **`begin="img1.beginEvent"`** specified by the SMIL element `img2`.

During the *boot reaction*, the program in Listing 3.6 evaluates the **`par/and`** in lines 1–22 (let's call it `par1`) and creates its two trails. The first (lines 2–13) has a **`par/or`** (`par2`) whose first trail just includes the player (line 4) i.e., it halts waiting for the event `start`, and the second trail (lines 6–12) creates a **`par/and`** composition (`par3`) whose both trails halts for `3s` (lines 7 and 10). The second trail of `par1` (lines 14–22) halts waiting for the event `started` (line 15). At this point, all activated trails are waiting and the *boot reaction* finishes.

At 3*s* of the logical time, the program wakes both trails of `par3`, executing them in order. The first emits the event `start` (line 8). This event wakes the player in line 4, which goes to *playing mode* to execute `<img1>` and emits the event `started`. The program then wakes from the **`await`** in line 15, evaluates the **`par/or`** in lines 17–21 (`par4`) and creates two trails: the first (line 18) executes `<img2>` and halts until that object finishes; and the second halts for 4*s* (line 20).

The reaction follows, and the second trail of `par3` executes and emits the event `stop` (line 11). This event wakes the player, which goes to *sleeping mode*, and the reaction finishes. At the end of this reaction, `par3` has ended because both of its trails have executed and terminated. From now on, the program,

```
1  par/and         /* par1 */
2  do              /* img id=img1 */
3   par/or do      /* par2 */
4     #include "player.ceu";
5   with
6     par/and do   /* par3 */
7       await 3s; /* begin=3s */
8       emit start (<img1>);
9     with
10      await 3s; /* end=3s */
11      emit stop;
12    end
13  end
14 with             /* img id=img2 */
15  await started;
16     /* begin=img1.beginEvent  */
17  par/or do      /* par4 */
18    await Play (<img2>);
19  with
20   await 4s;     /* dur="4s" */
21  end
22 end
23
```

```
1 <par id="par1">
2   <img id="img1" begin="3s"
3              end="3s".../>
4   <img id="img2" dur="4s"
5   begin="img1.beginEvent".../>
6 </par>
7
```

Listing 3.6: A Céu program handles internal events immediately when they are emitted following a stack-based processing.

which has only the second trail of `par1` active, is halted waiting for $4s$.

At $7s$ of the logical time (i.e., $4s$ after the last reaction) the program wakes from the `await` in line 20. As there is no other statement, that trail ends, aborting `par4`, therefore, the execution of `img2` finishes. Now both trails of `par1` have finished, which ends that composition, as well as the whole program.

Summing up, the program in Listing 3.6 executes at $3s$ `img1` for $0s$ (its starting and its ending occur within the same reaction) and `img2` for $4s$. There is no other possible execution for that code.

The discussion in this section illustrates how the Céu synchronous semantics prevents the programming of non-deterministic multimedia applications. However, because Céu is a general purpose language, it has no primitives for performing multimedia operations. Next chapter describes Céu-Media, a multimedia library for programming multimedia in Céu. Céu-Media is not just yet another multimedia framework that implements high-level functions for executing low-level operations. Céu-Media realizes the synchronous semantics of the language in the final multimedia application, guaranteeing not only the determinism, but also the precise intermedia synchronization.

# 4
# The Design and Implementation of Céu-Media

In this chapter we present Céu-Media, a practical contribution of this thesis. We first discuss in Section 4.1 the gains of having a multimedia engine that follows the synchronous semantics. Then, in Section 4.2 we introduce Céu-Media, its implementation and the code of three multimedia applications that use it. In Section 4.3 we present the low-level C multimedia backend used by Céu-Media. We close this chapter by discussing the scientific contributions of Céu-Media in Section 4.4.

## 4.1
## Synchronous Multimedia Engine

Figure 4.1 illustrates a generic workflow that most multimedia players implement (of course, each one with its own specificities). This workflow has two macro components: the Controller and the Multimedia Engine. The Controller maintains the logic and state of programs; and the Multimedia Engine synthesizes and synchronizes audio samples and video buffers, creating the Multimedia Output. The Controller uses the API of the Multimedia Engine to control (e.g., start, stop, pause etc.) objects in the Multimedia Output. The Multimedia Engine also notifies the controller about events occurred in the presentation (e.g., end of an object, input events, etc.).

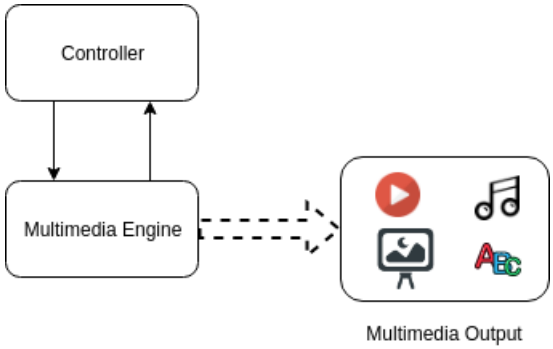


Figure 4.1: General architecture of multimedia players.

So far in this thesis we have mainly discussed issues related to the Controller component. Our claim is that the lack of deterministic semantics of traditional multimedia languages may lead to the development of players

(controllers) whose behavior cannot be fully predicted. And we argue in favor of controllers that follow the synchronous semantics, because this execution model prevents several problems, as discussed in Chapter 3.

However, using a synchronous language to program the Controller component solves just part of the problem of developing deterministic multimedia applications. A comprehensive approach for this problem should also address the Multimedia Engine component. To illustrate, consider the code in Listing 4.1. A CÉU programmer expects that the object `VID_1` in line 2 starts at the beginning of the program; the object `VID_2` in line 6 starts exactly $2s$ (logical time) after `VID_1`; and that the program presents only $5s$ of `VID_1` and $3s$ of `VID_2` (also logical time). In this example, the function `Play ()` is part of the Multimedia Engine API.

```
1 par/or do
2   await Play (<VID_1>);
3 with
4   await 2s;
5   < non-blocking statements (no await) >
6   await Play (<VID_2>);
7 with
8   await 5s;
9 end
```

Listing 4.1: The programmer expects that the Multimedia Ouput respects the synchronous semantics of CÉU.

In Chapter 1 we have highlighted that the definition of determinism should consider the timing aspect. Thus, in this simple example, if the Multimedia Engine follows the "physical" time, there is no guarantees that `VID_2` would start exactly after $2s$ of presentation of `VID_1` and the duration of both objects would be as expected. Furthermore, in multiple executions of this program the difference between the beginning of `VID_2` and `VID_1` can be different. That is, even though this program has been developed using a deterministic language, the Multimedia Output would be non-deterministic.

The approach we adopt to address this problem is to ensure that the Multimedia Engine also follows the synchronous semantics and the logical time to synthesize the presentation. In this case, as both components (Controller and Multimedia Engine) have the same execution model and timing reference, the operations issued by the Controller would be performed at the appropriate timing in the Multimedia Output, guaranteeing its determinism.

## 4.2 Céu-Media

Céu-Media[90] is a Multimedia Engine designed with two main goals: to provide high-level abstractions for programming multimedia in Céu; and, to realize the synchronous semantics in the Multimedia Output. The designing of its API has been inspired by the languages NCL and SMIL. Here we describe Céu-Media[1] internals and the programming model promoted by its API.

### 4.2.1 A Hello World in Céu-Media

Céu-Media API implements three abstractions: `Scene`, `Properties`, and `Player`. A `Scene` represents a top-level OS window. `Properties` define a map of key/values corresponding to different properties of each media object. And a `Player` renders an object on a `Scene` following a given set of `Properties`. Listing 4.2 depicts a simple Céu-Media application that uses these abstractions to present two videos side-by-side for 15s on screen, restarting them whenever both end.

```
1  var int width = 1080;
2  var int height = 720;
3  var [] byte uri = []."resources/animGar.mp4";
4  var Properties.Video prop1 = Properties.Video (
5                  Region(0, 0, width/2, height/2, 1), 1.0, 1.0);
6  var Properties.Video prop2 = Properties.Video (
7            Region(width/2, 0, width/2, height/2, 1), 1.0, 1.0);
8  var&? Scene scene = spawn Scene(Size(width,height));
9  watching  (scene) do
10   watching 15s do
11     loop do
12       par/and do
13         await Play (uri, prop1, scene);
14       with
15         await Play (uri, prop2, scene);
16       end
17     end
18   end
19 end
```

Listing 4.2: Two videos side-by-side in Céu-Media.

Lines 4–7 define two video `Properties` variables, `prop1` and `prop2`. The first defines that an object is to be played on the region delimited by the given rectangle (`Region (0,0,width/2,height/2,1)`) with its normal volume (`1.0`) and opacity (`1.0`). Similarly, the second defines that an object is to be played on another region (`Region (width/2,0,width/2,height/2,1)`) also with its

[1]The Céu-Media source code is publicly available at `http://rodrimc.github.io/ceu-media`.

normal volume and opacity. Note that these `Properties` declarations are only descriptions used by `Players` to determine how they should render a video on a `Scene`. Thus at this point (line 7) nothing has happened and the screen is empty—in fact, time has not even passed.

Line 8 creates a `Scene` with 1080x720 pixels and store it in variable `scene`. CÉU **`spawn`** statement executes code abstractions (CÉU mechanism for defining subprograms) in parallel with the caller code, returning a handle to that subprogram instance. The **`watching`** block in lines 9–19 aborts its execution when the `Scene` ends (either due to a normal end or due to an error). Next statement defines another **`watching`** block (lines 10–18). It defines an execution block with a duration of 15$s$, that is, a block that executes its body for at most 15 seconds, and terminates. Here the body (lines 11–17) consists of an infinite loop whose sole statement is a **`par/and`** composition (lines 12–16) with two execution trails, each also consisting of a single statement (line 13 and 15). Once executed, the **`par/and`** statement starts its trails in parallel and terminates only after both of them terminate. In this case, the first trail creates a `Player` to render the file pointed by the variable `uri`, with `Properties prop1` on `Scene scene`, starts it, and waits for its end. Similarly, the second trail creates another `Player` to render the same file, but according to `Properties prop2` on the same `Scene`, starts it, and waits for its end.

When the program in Listing 4.2 starts, the two players are created and start to render the same video in parallel. Whenever *both* of them end, the whole **`par/and`** statement terminates and is immediately restarted by the outermost loop, which means that new players are created and started. This process goes on until the 15th second is reached, at which point the inner **`watching`** block terminates. At this point, the outer **`watching`** block also terminates, because there is no other statement to execute, thus the whole program ends. Figure 4.2 presents a screenshot of this program.

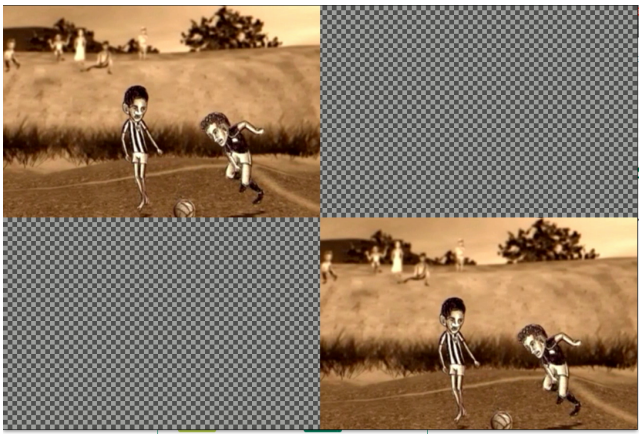

Figure 4.2: Screenshot of the execution of program in Listing 4.2

Note in the screenshot that both `Player`s are presenting exactly the same video frame. This is not a coincidence, but rather a feature enforced by Céu-Media synchronization model: it uses the program's logical time for synchronizing all objects in the presentation. Section 4.2.4 and Section 4.3 detail how Céu-Media achieves this frame-level synchronization precision.

### 4.2.2 Céu-Media Programming Model

Céu-Media programming model is based on the usual idea of multimedia scenes, which are responsible for composing and synchronizing the output of different multimedia objects into a unified presentation. Each `Scene` has an internal clock that rules the rendering synchronization. `Scene`s also provide a limited form of compositionality[2]: pausing and resuming a `Scene` pauses and resumes all of its objects; and destroying a `Scene` also destroys all of its objects.

`Player` is the Céu-Media abstraction that represents a multimedia object in a `Scene`. The main responsibility of a `Player` is to decode the object pointed by an URI and output a sequence of raw audio and/or video buffers. To create a `Player`, one has to pass a given set of `Properties`. Thus, the `Player` applies a sequence of low-level operations for ensuring that the output buffers match the values specified in that `Properties` set.

Figure 4.3 depicts a schematic illustration of these abstractions. In our current implementation, a `Scene` opens a OS-level window to render the presentation. Given this programming model, developing a multimedia application using Céu-Media becomes a matter of creating a `Scene` and starting a `Player` at the precise moment a multimedia object should be started. The API has functions to manipulate the properties of a `Player` while it is running, as well as functions to pause, resume and stop it.

### 4.2.3 Implementation

Under the hood, the `Scene`, `Player` and `Properties` abstractions are implemented using a mix of Céu and native code that invokes functions of the C multimedia library LibPlay[3] used as backend. However, Céu-Media exposes a pure Céu API, that is, users do not need to call any low-level native C function to use it.

[2]Limited compositionality because our current implementation does not support adding a scene to another scene. There are some engineering and researching challenges to address for implementing this feature. More on that in the conclusion chapter.

[3]https://github.com/TeleMidia/LibPlay

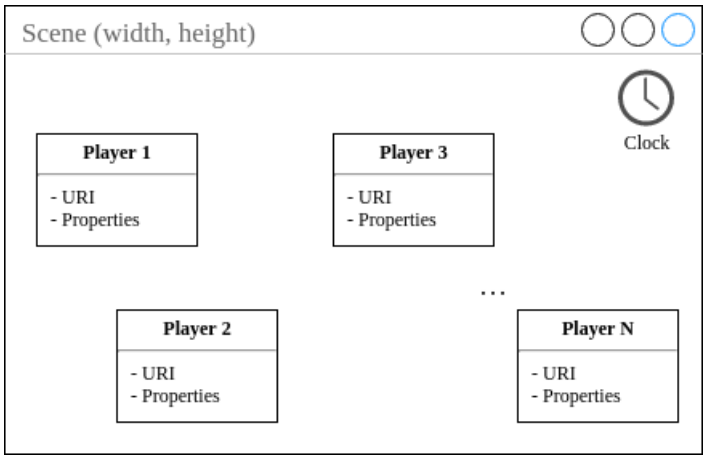


Figure 4.3: Schematic illustration of the abstractions implemented by Céu-Media.

### The `Properties` data type

The `Properties` type is a Céu tagged data type. Each tag groups properties related to one of the following media types: text, image, audio, or video. A simplified version of the Céu code that defines the `Properties` type is presented in Listing 4.3.

```
1  data Properties.Text with
2    var[] byte   text    = [] .. "";
3    var[] byte   font    = [] .. "";
4    var   Region region  = val Region (0, 0, 0, 0, 1);
5    var   uint   color   = 0xffffffff;
6  end
7
8  data Properties.Image with
9    var   Region region  = val Region (0, 0, 0, 0, 1);
10   var   real   alpha   = 1.0;
11 end
12
13 data Properties.Audio with
14   var   real volume    = 1.0;
15 end
16
17 data Properties.Video with
18   var   Region region  = val Region (0, 0, 0, 0, 1);
19   var   real   alpha   = 1.0;
20   var   r64    volume  = 1.0;
21 end
```

Listing 4.3: The `Properties` tagged data type.

A variable of type `Properties` holds a set of key/values, but has no behavior associated to it. Although more verbose, this design promotes reuse: different `Players` can share the same `Properties` description. It is somehow similar to the use of the `<descriptor>` element in NCL.

### The `Scene` code

A `Scene` is implemented as a CÉU code. Listing 4.4 depicts a simplified version of its execution body (lines 4–19).

```
1  code/await Scene (var Size? size)
2                -> (var& IScene handle)
3                    -> none
4  do                           /* body */
5    < create LibPlay objects >
6    finalize do
7      < clear LibPlay memory >
8    end
9    par/and do
10     loop do
11       evt = < get next event > ();
12       emit (evt);
13     end
14   with
15     every FREQ ms do
16       _advance_time (FREQ);
17     end
18   end
19 end
```

Listing 4.4: The `Scene` code.

When one defines a variable of type `Scene`, this code starts to run immediately: it executes in parallel with the surrounding code until the variable goes out of scope. The `Scene` body performs two main tasks: (i) it emits scene-level events to programs (mouse click, key press, etc); and (ii) it controls the scene clock. The `Scene` clock only advances through explicit calls to the function `advance_time` (line 16, in the previous listing). The inner workings of the scene clock and its impact on the synchronization of the Multimedia Output presentation are discussed in Section 4.2.4. Note the **`finalize`** block in the `Scene` body, it ensures that all allocated resources are properly cleared when the code finishes.

### The `Player` code

Each `Player` is another code that, when instantiated, immediately presents a media file according to a `Properties` description on the given `Scene`. When there is no more content to be presented (i.e., the `Player` has drained all of its media data), the `Player` stops (the code ends). Listing 4.5 depicts a sketch of the `Player` code. The function `Play` takes an `uri`[4], a `Properties`, and a `Scene` and returns a new `Player`. As in the `Scene` code, the `Player` **`finalize`** block guarantees that the LibPlay player stops whenever the corresponding `Player` variable goes out of scope, and that allocated resources are properly released.

[4] Even though our `Player` implementation takes an `uri` as input and supports streaming, the execution of remote objects rises issues that we do not tackle in this thesis. Thus, here

```
code/await Play (var& []byte uri, var& Properties prop, var& IScene scene)
                  -> (var& IPlayer handle)
                      -> none
do                          /* body */
  p = < LibPlay player  >;
  finalize
    _start (p);
  with
    _stop (p);
  end
  await p;
end
```

Listing 4.5: The `Player` code.

### 4.2.4 Synchronization: realizing the synchronous semantics in the Multimedia Output

Every `Scene` has an internal monotonic clock that rules the Multimedia Output. This clock starts with 0 and advances only through explicit calls to the function `advance_time()`. Such calls are triggered by the scene code itself (i.e., Céu-Media users should not worry about calling this function). For instance, in Listing 4.4, the `Scene` advances its clock every `FREQ` milliseconds (lines 15–17), where `FREQ` is an internal constant, by the corresponding amount of time. This call binds the logical time events of Céu with the "physical" clock used to synchronize all players in a scene as follows. A buffer generated by a `Player` has a presentation timestamp ($PTS$) and a duration ($dur$), it is rendered when the `Scene` clock matches its $PTS$, and it is presented respecting the $dur$ value. To illustrate the consequence of this binding of logical and physical time, consider the program depicted in Listing 4.6.

```
var&? Scene s = spawn Scene (Size (width, height));
var Properties.Video prop1 = Properties.Video (<...>, 0.0);
var Properties.Video prop2 = Properties.Video (<...>, 0.0);
var Properties.Video prop3 = Properties.Video (<...>, 0.0);
var Properties.Video prop4 = Properties.Video (<...>, 0.0);
var Properties.Audio prop5 = Properties.Audio (<...>, 1.0);

await 5s;
watching (s) do
  spawn Play("video.ogv", prop1, s);
  spawn Play("video.ogv", prop2, s);
  spawn Play("video.ogv", prop3, s);
  spawn Play("video.ogv", prop4, s);
  await Play("audio.oga", prop5, s);
end
```

Listing 4.6: Binding logical and physical time.

we consider that all media objects are locally available upon program execution.

The program first spawns a `Scene` in line 1, which immediately runs in parallel the code in Listing 4.4. The `Scene` code creates its internal objects and then starts to react to each `FREQ` timing events for advancing its clock. It worth remembering that the Céu compiler produces a single-threaded program, which means that there is no way the `Scene` advances its clock while the program is running another piece of code. In other words, during a reaction Céu-Media guarantees that the `Scene` clock remains the same.

The program proceeds and creates five `Properties`, four for videos and one for audio (lines 2–6). The program then waits for five seconds (line 8) and creates a **`watching`** block whose body spawns four `Players` (lines 10–13), initializing each with the same video URI and a corresponding `Properties` set; these are started as soon as they are created. Finally, it creates a `Player` (line 14) to play the audio, starts it, and waits for its end (`stop` event).

The only instructions that actually take time in this program are the **`await`** statements in lines 8 and 14, and the code that advances the scene clock (Listing 4.4, lines 15–17)—and they all consume exactly the specified amount of logical time. This means that logical time does not pass while the players are being spawned and started. Moreover, because the logical clock drives the physical (scene) clock, this also means that no samples are timestamped with distinct values during this time. Because the physical time actually passes while the program creates the players, without this precise control over the scene clock, each `Player` would set a different timestamp value on the produced samples. This would happen even though they have been created in the same reaction.

From the above we can say that the program in Listing 4.6 produces a Multimedia Output that renders four videos and an audio track in-sync.

### 4.2.5 Céu-Media Sample Applications

In this section we discuss the implementation of three different multimedia applications using Céu-Media. But before that, we introduce some terminology. Thinking in terms of modeling concepts and their relative level of abstraction, we regard the process of writing a multimedia application in Céu-Media as consisting of four layers, depicted in Figure 4.4.

Layer 0 is the base layer; it is a C API for programming multimedia. (LibPlay). Layer 1 is Céu-Media itself; it is written in Céu upon Layer 0, hides its complexity, and exposes to the upper layer a pure high-level Céu API (the `Properties` types and the `Scene` and `Player` codes). Layer 2 consists of Céu-Media applications, i.e., Céu programs that use the Céu-Media

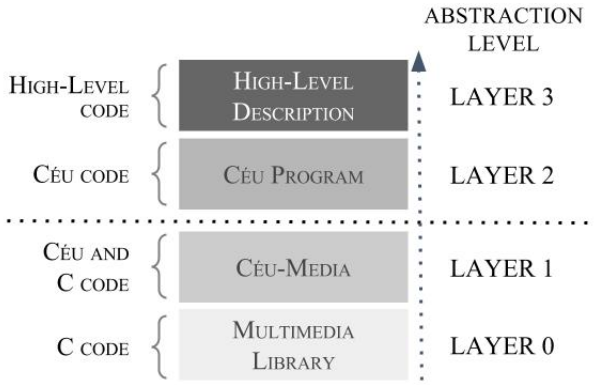


Figure 4.4: Abstraction layers when programming applications in CÉU-MEDIA.

extensions for developing multimedia presentations. One could stop in Layer 2, but it is possible to go further. Using CÉU mechanisms we can combine the basic abstractions of CÉU with those of CÉU-MEDIA into novel abstractions that are better suited to the description of particular scenarios. For instance, below we discuss the definition of a abstraction for constructing multimedia slideshows. These CÉU-MEDIA extensions appear in Layer 3, the uppermost layer in terms of level of abstraction. Another possibility would be to implement an NCL or SMIL player using CÉU-MEDIA. In this scenario, the player would fit in Layer 2, and high-level declarative presentations written in NCL or SMIL would fit in Layer 3. From now on, whenever a code listing is presented, we indicate its position in this abstraction scale.

### A SRT Reader

Listing 4.7 depicts the partial CÉU code that renders `SRT` subtitles. When instantiated, it reads a SubRip text file and, for each subtitle entry, obtains its start time, end time, and text (lines 5–7), awaits for the amount of time corresponding to its start time (line 8), and creates a `Player` that renders the subtitle text for the duration of the entry (lines 9–13) .

```
1  code/await SRT (var& Scene scene, var[] byte file, var int y_offset) -> none
2  do
3    var int now = 0;
4    loop entry in < subtitle entries in file > do
5      var int from = get_start_time (entry);
6      var int to = get_end_time (entry);
7      var[] byte text = get_subtitle_text (entry);
8      await (from - now)ms;
9      watching (to - from)ms do
10       var Properties.Text text = Properties.Text (text, [] .. "sans 40",
11                     Region(0, y_offset, 800, 100, 1), 0xffff0000);
12       await Play(_, text, scene);
13     end
14     now = to;
15   end
```

```
16 end
```

Listing 4.7: The SRT organism (Layers 1–2).

Listing 4.8 depicts a code excerpt that uses the SRT code for executing a video with its subtitle.

```
1  var[] byte video    = < video URI >;
2  var[] byte subtitle = < subtitle URI>;
3
4  var int width = 1080;
5  var int height = 720;
6
7  var Properties.Video p = Properties.Video (
8    Region(0, 0, width, height, 1), 1.0, 1.0);
9
10 var&? Scene scene = spawn Scene(Size(width,height));
11 watching (scene) do
12   par/and do
13     await Play (video, p, scene);
14   with
15     await SRT (scene, subtitle, 650);
16   end
17 end
```

Listing 4.8: Playing a video with subtitles (Layer 2).

The complete implementation of the SRT code demands the use of asynchronous I/O operations for reading the SRT file, along with **await** statements for synchronizing the asynchronous calls, as the use of traditional blocking I/O would violate the synchronous hypothesis. Thus a programmer writing this code needs to work on Layers 1 (asynchronous I/O) and 2 (text rendering via Céu-Media). However, using this abstraction to render subtitles does not require writing any C code neither synchronizing asynchronous I/O operations, that is, programmers write code that fits in Layer 2, as illustrated in Listing 4.8.

In the Céu-Media repository one can find the complete implementation of this code using the asynchronous I/O library Céu-libuv[5] (a wrapper for the C library LibUV[6]).

### A Multimedia Slideshow

The slideshow we consider consists of three images. Each one is presented for five seconds while a piano soundtrack is played in background (in a loop) and synchronized subtitles are shown over the images. The slideshow terminates when all three images are displayed or when there are no more

[5] https://github.com/fsantanna/ceu-libuv
[6] https://github.com/libuv/libuv

subtitles to be presented or any key is pressed. Listing 4.9 depicts the Céu-Media code of this application.

```
1  var int width = 800;
2  var int height = 585;
3
4  var Properties.Image p1 = Properties.Image (
5        Region (0,0,width,height,1), 1.0);
6  var Properties.Audio p2 = Properties.Audio (0.5);
7
8  var&? Scene scene = spawn Scene(Size(width,height));
9  watching (scene) do
10   par/or do
11     loop do
12       await Play ("piano.oga", &p2, &scene);
13     end
14   with
15     watching 5s do
16       await Play ("img1.jpg", &p1, &scene);
17     end
18     watching 5s do
19       await Play ("img2.jpg", &p1, &scene);
20     end
21     watching 5s do
22       await Play ("img3.jpg", &p1, &scene);
23     end
24   with
25     await SRT (&scene, "subtitle.srt", 485);
26   with
27     await CM_SCENE_KEY;
28   end
29 end
```

Listing 4.9: A multimedia slideshow (Layer 2).

The previous `par/or` composition (lines 10–28) and the sequence of `watching` statements (lines 15–23) resemble the `par` (with its `endsync` attribute equals to `first`) and `seq` SMIL containers. The `watching` blocks resemble SMIL's `dur` attribute, while the counterpart of the previous `loop` statement (lines 11–13) is the `repeatCount` attribute of SMIL, with its value set to `indefinite`. Similar analogies can be made with NCL. But the crucial difference here is that the semantics of Céu is unambiguous and guarantees that the trails are, at any time, precisely and deterministically synchronized.

### A Multimedia Slideshow "Reader"

The code that implements the slideshow semantics can be encapsulated in a Céu code abstraction. Next example goes in this direction, but it generalizes the Slideshow code to read from a Lua table some parameters to be used in the presentation. Here we chose Lua for mere convenience—Céu integrates seamlessly with Lua: codes within tokens `[[` and `]]` are executed by the Lua

interpreter. Any higher-level syntax could be used, provided that there is a corresponding CÉU parser to it.

As an example, consider the Lua table depicted in Listing 4.10, it defines the `width` and `height` of a slideshow presentation, as well as the `time` each image should be presented. The table also defines the uris of the images, background audio and the subtitle to be presented.

```
SLIDESHOW = {
  width = 800,
  height = 585,
  time = 5,
  audio = "piano.oga",
  subtitle = { uri = "subtitle.srt", y_offset = 485},
  images = { "img1.jpg", "img2.jpg", "img3.jpg" }
}
```

Listing 4.10: A Lua table defining some parameters of a slideshow (Layer 3).

Listing 4.11 depicts the source code of a Slideshow Reader. It assumes a global Lua table called `SLIDESHOW` that sets the parameters of the slideshow.

```
code/await Slideshow_Reader (none) -> none
do
  var int width = [[ SLIDESHOW.width ]];
  var int height = [[ SLIDESHOW.height ]];
  var int dur = [[ SLIDESHOW.time ]];

  var Properties.Image p1 = val Properties.Image (
    Region (0, 0, width, height, 1), 1.0);
  var Properties.Audio p2 = val Properties.Audio (0.5);

  var&? Scene scene = spawn Scene(Size (width, height));
  watching (scene) do
    par/or do
      var[] byte audio = [[ SLIDESHOW.audio ]];
      loop do
        await Play (audio, p2, scene);
      end
    with
      var ssize n = [[ #SLIDESHOW.images ]];
      var ssize i;
      loop i in [1 -> n] do
        var[] byte uri = [[ SLIDESHOW.images[@i] ]];
        watching (dur)s do
          await Play (uri, p1, scene);
        end
      end
    with
      var[] byte uri = [[ SLIDESHOW.subtitle.uri ]];
      var int y_offset = [[ SLIDESHOW.subtitle.y_offset ]];
      await SRT (&scene, uri, y_offset);
    with
      await CM_SCENE_KEY;
    end
  end
end
```

Listing 4.11: A slideshow code abstraction that reads some parameters from a Lua table (Layer 2).

It worth mentioning that the codes in Listing 4.9 and Listing 4.10 are equivalent, i.e., they produce the same Multimedia Output, but in the latter case the objects of the presentation were specified declaratively without any Céu code. The point of this example is to illustrate how from a small set of abstractions exposed by Céu-Media one can create higher-level constructs targeting nonspecialist (regarding Céu and multimedia) users. Such usage resembles the use of template languages such as TAL [91] or XTemplate [92] in the domain of XML languages.

### A TV Controller

The last example discussed in this chapter regards the implementation of a TV-like controller that allows users to choose a video to watch from a set of five videos. At any time, users can switch videos and increase or decrease the audio volume.

Listing 4.12 depicts the code of this controller. The set of videos is defined through a Lua table in lines 8–12. This example illustrates that using Céu internal events, one can decouple the handling of external events with the application logic. The code below uses four internal events: `previous_video`, `next_video`, `increase_volume` and `decrease_volume` (lines 1–4). The second trail of the **`par/and`** implements the logic of the controller itself: it reacts to each of those internal events for updating its internal state (lines 14–55). Céu-Media `Player_Set_Int ()` function takes as argument a handler for a `Player` instance, the name of a property and an **`int`** value and sets this value to that property of that `Player` (lines 39 and 37).

```
1  event (none) previous_video;
2  event (none) next_video;
3  event (none) increase_volume;
4  event (none) decrease_volume;
5
6  [[
7    videos = {
8      "vid1.ogv", "vid2,ogv", "vid3.ogv", "vid4.ogv", "vid5.ogv"
9    }
10 ]]
11
12 par do
13   #include "inputs.ceu";
14 with
15   var&? Scene scene = spawn Scene (1080, 720);
16   watching (scene) do
17     var real vol_level = 1.0;
18     var int = 1;
```

```
19      loop do
20        var [] byte uri = [[ videos[@i] ]];
21        var Properties.Video p = Properties.Video (
22          Region (0, 0, 1080, 720, 1), 1.,0, vol_level);
23
24        var&? Play player = spawn Play (uri, &p, &scene);
25        par/or do
26          every increase_volume do
27            if vol_level < 1.0 then
28              vol_level = vol_level + 0.1;
29            end
30            call Player_Set_Int (player, "volume", vol_level);
31          end
32        with
33          every decrease_volume do
34            if vol_level > 0.0 then
35              vol_level = vol_level - 0.1;
36            end
37            call Player_Set_Int (player, "volume", vol_level);
38          end
39        with
40          await next_video;
41          i = i + 1;
42          if i == 6 then
43            i = 1;
44          end
45        with
46          await previous_video;
47          i = i - 1;
48          if i == 0 then
49            i = 5;
50          end
51        with
52          await CM_PLAYER_STOP;
53        end
54      end
55    end
56  end
```

Listing 4.12: A TV-like controller in Céu. (Layer 2).

The first trail of the **par/and** in the code above includes the file `inputs.ceu`, which is responsible for handling external input events and emitting the internal events the controller expects. Listing 4.13 illustrates a possible input handler for this controller. It reacts to each occurrence of the Céu-Media `CM_SCENE_KEY` event and emits an appropriate internal event if the key pressed is one of the keyboard arrow keys.

```
1  var uint scene_id;
2  var [] byte key;
3  var bool is_pressed;
4
5  every (scene_id, key, is_pressed) in CM_SCENE_KEY do
6    if is_pressed then
7      if key == "LEFT" then
8        emit next_video();
9      else/if key == "RIGHT" then
10       emit previous_video();
```

```
11      else/if key == "UP" then
12        emit increase_volume();
13      else/if key == "DOWN" then
14        emit decrease_volume();
15      end
16    end
17  end
```

Listing 4.13: An input handler for the TV-like controller. (Layer 2).

## 4.3 The low-level Multimedia Backend

LibPlay, the supporting C library Céu-Media uses, is a wrapper to the industry-grade multimedia framework GStreamer. It has been designed to hide part of the complexity of GStreamer through a simple and high-level API. In this section we present the most relevant aspects of both GStreamer and LibPlay to describe how we managed to reproduce the synchronous model in the final multimedia output.

### GStreamer and LibPlay

GStreamer is an open source framework that supports the development of applications that process multimedia content (e.g., media players, video editors, transcoders, media streamers, and so on). It has a modular, flexible and plugin-oriented architecture, and runs on all current major operating systems.

The framework adopts a pipeline-based programming model: processing elements are connected in an acyclic and directed graph to process multimedia data. Figure 4.5 illustrates a typical GStreamer pipeline for rendering an Ogg file—Ogg is a format for multiplexing audio, video and text content into a single container. In the figure, vertices are elements that process the data and edges connect the output of an element to the input of other.

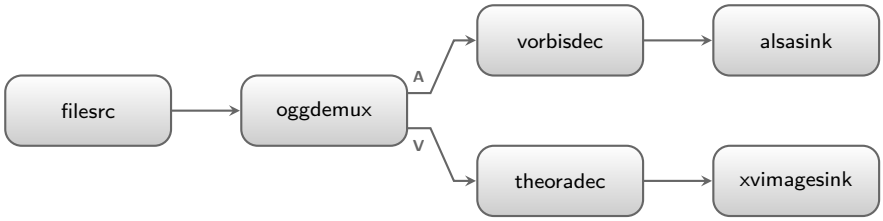


Figure 4.5: A GStreamer pipeline that plays an Ogg file [3].

Elements that have no incoming edges are called *source*, that is, they are data producers. In Figure 4.5, the *filesrc* is the only source element and is

responsible for reading an Ogg file and outputting a multiplexed byte stream having its content. The next element in the graph is the *oggdemux*, which receives as input that byte stream, demultiplexes it, and produces two coded byte streams, one for audio and other for video. This pipeline assumes that the Ogg file has just a single audio and a single video streams, encoded using the Vorbis and Theora codecs, respectively. The *vorbisdec* and *theoradec* are the elements that decode the content and output raw audio samples and video buffers. Elements that have no outcoming edges are called *sinks*. Our example has two sinks: *alsasink* and *xvimagesink*. Sink elements are the final data consumers in a pipeline and, in general, responsible for the actual rendering. Thus, the *alsasink* element receives raw audio samples and reproduces them in the sound card, while the *xvimagesink* receives raw video buffers and opens an window for rendering the video.

This pipeline-based programming model is very flexible. For instance, to adapt the example above to play another file format it is enough to replace the demuxer and decoder elements to others more suitable to that file. Thus, roughly speaking, programming multimedia applications using GStreamer involves instantiating adequate elements and properly linking them for processing the media data flow.

The framework is also able to provide fine-grained synchronization of different media streams. Pipelines have a clock (*GstClock*) that monotonically returns an absolute time, which is used for synchronizing the output. Each audio sample and video buffer have a *PTS* (Presentation timestamp) and *dur* (duration) fields. The synchronization procedure is usually executed by sink elements following this general idea: samples/buffers received before their presentation time are buffered; and samples/buffers received after their presentation time are discarded. Then, sinks guarantee that each buffered data is presented when the presentation time (ruled by the internal pipeline *GstClock*) matches their PTS, and also that samples/buffers are presented only for their appropriate durations.

The GStreamer flexibility comes with the price of complexity: programming some usual operations (e.g., dynamically change a running pipeline, pausing a single media stream, etc.) can be very complex. Furthermore, the extensive use of callbacks for event handling requires that programmers design their codes protecting shared variables from concurrent access. Thus, to overcome these issues and other idiosyncrasies, we have developed LibPlay.

LibPlay is a multimedia library built on top of GStreamer that tries to hide most of the complexity of that framework through a multimedia scene-based API. This library has three abstractions: scene, media and events.

LibPlay programming model is similar to Céu-Media: a scene composes and synchronizes multiple multimedia objects (called media) and interacts with applications through events.

A LibPlay scene (*lp_Scene*) manages a GStreamer pipeline that have mixers and sinks for audio and video. Figure 4.6 depicts an overview of this pipeline. Each Media Bin in the figure represents a LibPlay media, which can produce a stream of video or audio (or both). The element *compositor* is the video mixer, which is responsible for receiving the video output of each media bin and composing them into a single video stream. Similarly, the element *audiomix* receives audio outputs of media bins and composes them into a single audio stream. The final video and audio streams are then received by the respective sink elements (*xvimagesink* and *alsasink*) for rendering.

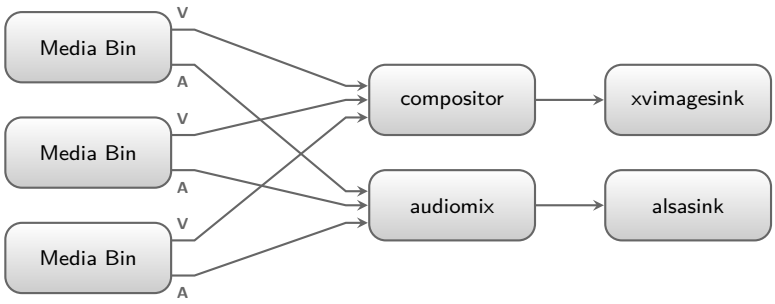


Figure 4.6: An overview of a LibPlay pipeline.

A LibPlay media (*lb_Media*) is implemented as a GStreamer *bin*. A bin is a generic container that groups linked elements into one logical element that can be added to a pipeline. Figure 4.7 illustrates an overview of a media bin. The *uridecodebin* source element is responsible for opening a file pointed by an URI and decoding it. Depending on the file content, this element can produce different media streams. For instance, decoding a typical video file outputs a video and an audio stream, and possibly a text stream if it has embedded subtitles. Each stream then goes to an appropriate sequence of filters that applies operations (e.g., cropping, scaling, changing volume, changing transparency, etc.) according to the properties set of that media. Finally, the transformed stream goes either to scene's *compositor* or *audiomix* for being composed with other streams before the actual rendering.

As usual, the LibPlay scene's pipeline has a clock for synchronization. But the library implements its own clock (*lp_Clock*) for fine-grained control over the pipeline time. The *lp_Clock* has two operations mode: *normal* and $lock-step$. When operating under the *normal* mode, the clock time increases following the "physical" clock. But when operating under the $lock-step$

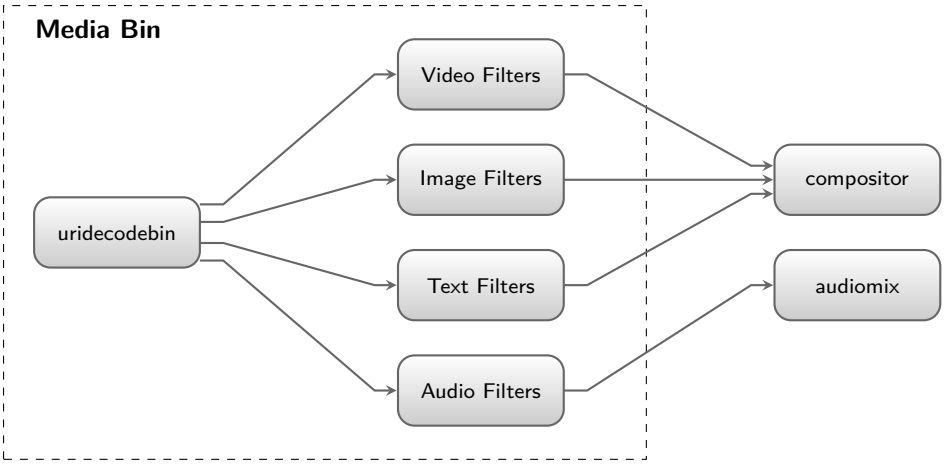


Figure 4.7: An overview of a LibPlay Media.

mode, the internal clock time only changes through a call to the function `lp_clock_advance()`.

Listing 4.14 illustrates the difference among the operation modes. In the left-side code, the *lp_Clock* is instantiated in line 2 under the *normal* mode. The *assert*() in line 5 may fail or not: if the running machine is fast enough, the physical time difference between the execution of lines 2 and 5 may be smaller than the clock time precision, then the *assert*() passes; otherwise it fails. The *sleep*(2) in line 7 suspends the execution of the program for $2s$. However, according to the POSIX.1-2017 specification, "*the suspension time may be longer than requested due to the scheduling of other activity by the system*"[93]. Thus, the *assert*() in line 11 may also fail depending on the system scheduling.

```
1  /* normal mode */
2  lp_Clock *c = lp_clock_new (NORMAL);
3
4  /* this assert may fail */
5  assert (lp_clock_get_time(c) == 0);
6
7  sleep (2);
8
9
10 /* this assert may fail */
11 assert (lp_clock_get_time(c) == 2000);
12
```

```
1  /* lock-step mode */
2  lp_Clock *c = lp_clock_new (LOCKSTEP);
3
4  /* this assert never fails */
5  assert (lp_clock_get_time(c) == 0);
6
7  sleep (2);
8  lp_clock_advance (c, 2000);
9
10 /* this assert never fails */
11 assert (lp_clock_get_time(c) == 2000);
12
```

Listing 4.14: When a *lp_Clock* operates under the lock-step mode, users has a fine-grained control over its time.

When the clock operates in $lock - step$ mode the user controls how its time advances. Thus, in the right-side code of Listing 4.14, the *assert*() calls never fail. Because there is no call to the function *lp_clock_advance*() between the clock instantiation in line 2 and the *assert*() in line 5, the clock is guarantee to remain in $0s$. And regardless for how long the program remains suspended

in the $sleep(2)$ call (line 7) when it wakes it advances the clock exactlly $2000ms$ (line 8). Therefore, the second $assert()$ also passes.

### Ceu-Media and LibPlay

In CÉU-MEDIA implementation, we have used LibPlay as follows. A CÉU-MEDIA `Scene` internally creates a $lp\_Scene$ and sets its clock to operate in $lock-step$ mode. Similarly, a CÉU-MEDIA `Player` instantiates a $lp\_Media$ and adds it to the `Scene`'s $lp\_Scene$. And when the $lp\_Scene$ notifies the CÉU-MEDIA `Scene` that a LibPlay event has occurred, that event is transformed to a `Scene`- or `Player`-level event, that is then passed to the application as a CÉU input event.

## 4.4 Discussion

Even though its development has demanded a substantial engineering work, CÉU-MEDIA is not just yet another multimedia library. It is another evidence that the synchronous hypothesis might be an adequate solution to semantical problems of traditional multimedia languages. For instance, developing a SMIL or NCL player in CÉU using CÉU-MEDIA would indirectly "fix" the non-determinism problem in that particular implementation.

The implementation of CÉU-MEDIA indicates the feasibility of realizing the synchronous semantics in the final Multimedia Output. But, if for one hand the approach of enslaving the presentation clock to the logical time is crucial to our solution, for the other hand it leads to some glitches in the presentation, notably for aural media. The larger the skew between real time and logical time, the more noticeable are these glitches. An explanation for this issue is the way most sound cards work: they operate at high values of sampling rates and expect that an audio sample is available at each cicle. A discontinuity in this process leads to audio imperfections (glitches).

CÉU-MEDIA is also an evidence that CÉU is a suitable alternative for programming multimedia. The language was originally designed for the domain of wireless sensor networks and since then has been applied to different domains. Thus, the development of CÉU-MEDIA and its applications without any "hack" in the language (that is, using plain CÉU) indicates that CÉU constructs and abstractions can be used in different fields than those originally glimpsed by the language designers.

Last, this work points that the synchronous execution model is a good fit for developing multimedia engines. In this sense, the use of CÉU and CÉU-

MEDIA is a comprehensive proposal that uses the synchronous hypothesis in both, the Controller and the Multimedia Engine components to produce deterministic applications.

CÉU-MEDIA, however, has been designed without any support for distributed multimedia applications. In such scenarios, the synchronous hypothesis does not hold due to communication latency. MARS is a middleware that uses CÉU-MEDIA as Multimedia Engine, but it implements a set of functionalities for approaching distributed applications. The design, implementation and rationale behind MARS is discussed in the next chapter.

# 5
# Mars: GALS Middleware for Programming Distributed Interactive Multimedia Applications

As discussed in Chapter 2, deploying a synchronous distributed system in a network without bounded communication delay and/or jitter rises several issues. In particular, the one we are interested in is *consistency*. More precisely, our concern is to guarantee that all processes in a distributed multimedia application have the same global view of the system, that is, they all agree upon the order of events, including the timing they occurred.

In this thesis we investigate the GALS architecture for enforcing consistency. This chapter describes the results of this investigation, whose practical result is the Mars middleware. Mars supports the development of distributed multimedia applications using Céu and Céu-Media following the GALS design. It has a centralized architecture, whose central component, known as Mars server, plays a key role for guaranteeing the consistency of the distributed system.

We first briefly discuss the problem of consistency in distributed systems, and then we frame this problem to the domain of distributed interactive multimedia applications in Section 5.1. Then, we give an overview of Mars in Section 5.2, which is deepened when we present it by example in Section 5.3. Section 5.4 details how the middleware executes applications. Section 5.5 describes how we have approached the consistency problem in Mars. Section 5.6 presents the internal components of the Mars middleware. Section 5.7 discusses the compilation of a Mars application. Section 5.9 presents some samples applications developed using this middleware. And Section 5.10 discusses some points covered in this chapter.

## 5.1
## Consistency in Distributed Systems

A well known problem in the distributed systems field is that of guaranteeing consistency. To illustrate, consider Listing 5.1 (adapted from Lamport's work [4]).

Suppose processes $A$ and $B$ run concurrently on a shared-data distributed system. When process $A$ begins, it sets variable $a$ to 0. Likewise, process $B$

```
1  /* Process A */
2  a = 0
3  <...>
4  a = 1;
5  if (b == 0)
6  {
7    /* <...> critical section */
8    a = 0;
9  }
10
```

```
1  /* Process B */
2  b = 0;
3  <...>
4  b = 1;
5  if (a == 0)
6  {
7    /* <...> critical section */
8    b = 0;
9  }
10
```

Listing 5.1: If the system does not guarantee consistent access to shared variables, processes A and B might be both in their critical section at the same time [4].

sets variable $b$ to 0 at the beginning of its execution. Both processes have a critical section, and our concern is to devise a simple protocol for preventing they execute such a section at the same time.

The protocol works as follows. Process $A$ sets variable $a$ to 1, checks if the value of variable $b$ is 0, and, if so, $A$ enters in its critical section. Process $B$ executes similarly: it sets variable $b$ to 1, and checks whether the value of variable $a$ is 0 before entering in its critical section.

Let's call $w(x, y)$ the operation that assigns the value $y$ to variable $x$ and $r(x)$ the operation that reads and returns the value stored in variable $x$. If processes $A$ and $B$ run concurrently on different machines or on different cores of the same machine, writing operations issued by a process take some time (likely non-negligible) to be perceived by the other. Let's say that both processes have executed lines 4 and 5 in parallel nearly at the same time. Then, we have the situation depicted in Listing 5.2.

```
1  /* Process A */
2  t0: w(a, 1)
3  t1: r(b) //returns 0
4
```

```
1  /* Process B */
2  t0: w(b, 1)
3  t1: r(a) //returns 0
4
```

Listing 5.2: Operations issued concurrently by processes $A$ and $B$.

At $t0$, process $A$ issues the operation $w(a, 1)$ and, at the same time, $B$ issues the operation $w(b, 1)$. Before these writings have taken effect to the other process, each one executes a reading operation. At $t1$, process $A$ does a $r(b)$, which returns 0. Also at $t1$, process $B$ does a $r(b)$, which returns 0 too. This may happen, for instance, due to network latency or cache coherence problems. In other words, process $A$ sees the following sequence of operations: $w(a, 1), r(b), w(b, 1)$; and process $B$ sees this sequence: $w(b, 1), r(a), w(a, 1)$.

The research community has been historically tackling this problem by means of defining *consistency models*. Each one has its advantages and

drawbacks, and the choice of which model to implement should consider the peculiarities of the applications a given system has to support. Our work is based on the sequential consistency model and adds a timing restriction to it.

### Sequential Consistency

Proposed by Lamport [4], the *sequential consistency* model provides total ordering of messages. According to this model, a system is said consistent if the following conditions are met [4]:

- the result of any execution is the same as if the operations of all processes were executed in *some* sequential order;
- the operations of each individual process appears in this sequence in the same order it has executed them.

The first condition defines that all processes agree upon the same ordering of events, which is not necessarily the same order seen by an outside observer that can timestamp each operation using a perfectly synchronized wall-clock. The second condition states that the order of operations issued by any particular process should be maintained in the global ordering.

Under the sequential consistency model, in each execution of the system all processes always see the same order of operations, but in successive runs such ordering may change due to the non-deterministic communication delay. If one can somehow enforce the same delay in multiple executions, then successive runs hold the same global ordering.

### Other consistency models

There are other models proposed in literature. The *strict* consistency is the strongest model proposed. It states that all writing operations should be immediately seen by all processes. This model is supported in single-core architectures and generally impossible to implement in multi-core or distributed systems.

The *causal* consistency is a weaker model than the sequential and provides a partial ordering of messages. It enforces that just causal-related operations should be executed in the same order. Thus, if operation $A$ is said to cause operation $B$, then all processes should execute $A$ before $B$. Any other operation that is not causally related can be executed in any order.

Another important model is the *eventual* consistency, that states that the value of a specific data will eventually converge in all processes given enough time without updates in the system.

### Consistency in Distributed Interactive Multimedia Applications

In essence, consistency models define how operations should be propagated throughout a distributed system and the properties the system should guarantee to its supported applications. In a distributed interactive multimedia application, such operations may be higher-level events, such as `START`, `STOP`, `PAUSE`, etc., instead of writings and readings to variables in traditional shared-memory distributed systems.

In this work, as described in Chapter 1, our focus is on distributed applications characterized by cooperating processes running on different devices. In these applications, control-based and collaboration-based communication are the most common types of interactions.

To properly support the development of these applications, the underlying system should guarantee that all processes have the same view of the order of events, otherwise the control or collaboration is harder to achieve. Consider, for instance, a LAN-based distributed game in which each player uses his/her personal device for controlling the corresponding avatar, and the composition of all interactions is presented on a TV. If different devices see different order of events, it is likely that some glitches may happen during the game, like the dead-man shoots situation [94]: an avatar that is dead for some players, but lives for others, shoots another avatar.

Consider now a scenario of two remote controls and multiple TV sets. Assume that each of these remote controls may change the channel of all TVs. If users interact concurrently with both controls and the system does not guarantee that all TVs see the same order of events, at the end of the interaction each TV may be on a different channel.

In this chapter we describe how we have tackled this consistency problem in distributed interactive multimedia applications.

## 5.2 Mars in a Nutshell

When one moves from local to distributed applications, the properties that Céu-Media guarantees cannot be maintained due to the violation of the synchronous hypothesis. This led us to investigate an approach for developing distributed multimedia systems whose processes rely on Céu-Media

and communicate asynchronously through a network without communication guarantees. This scenario typically suggests the use of the GALS architectural style.

The Mars middleware is part of the result of this investigation. It has a centralized architecture, it supports the communication of processes running on different devices and it implements the timing-sequential consistency model. In a Mars distributed application there is no notion of global synchronized clock or assumption regarding maximum communication latency. Each process runs synchronously and may emit asynchronous events that are received by others.

The middleware guarantees two properties: i) processes receive events in the same order; and ii) processes receive all events at the same logical time (different from the logical time at which they were emitted). Thus, Mars guarantees that all nodes agree not only upon the global ordering of events (property *i*) but also upon their timing (property *ii*).

Furthermore, the programming model enforced by Mars promotes the decoupling between the application logic and the specification of how devices communicate one another. That is, the application logic has no explicit communication primitives, but rather inter-application communication bindings are defined by an external script. Next section presents Mars in more details.

## 5.3 Mars by Example

We introduce Mars through a practical example. Let's take again the TV Controller sample application discussed in Section 4.2.5 and modify it to run on a distributed setting. For didactic purposes, first we consider the scenario of only two devices connected to the same LAN. One of them runs a program that handles user inputs (remote control) and the other runs another program that renders the videos (TV). When one presses one of the remote control arrow keys, the TV should respond as follows: Left and Right should replace the current video with the previous or next one, respectively; Up and Down should increase or decrease the audio volume, respectively.

The local version of this application has two source files, one for implementing the TV controller logic (Listing 4.12) and another one for handling input events (Listing 4.13). In the distributed version, each of these source codes generates a standalone program that is executed on the appropriate device. In essence, both source codes are the same discussed in last chapter, with a slight change: in the distributed version we use Céu external events instead of internals, because the latter type serves only for communication among trails within the same program. Listing 5.3 depicts a shortened version of these codes

using Céu external events.

```
input (none) PREVIOUS_VIDEO;
input (none) NEXT_VIDEO;
input (none) INCREASE_VOLUME;
input (none) DECREASE_VOLUME;

< ... >

var&? Scene s = spawn Scene (<...>);
watching (scene) do
  loop do
    var [] byte uri = <...>;

    var&? Play player =
                spawn Play (<...>);
    par/or do
      every INCREASE_VOLUME do
        < ... >
      end
    with
      every DECREASE_VOLUME do
        < ... >
      end
    with
      await NEXT_VIDEO;
      < ... >
    with
      await PREVIOUS_VIDEO;
      < ... >
    with
      await CM_PLAYER_STOP;
    end
  end
end
```

```
output (none) LEFT_KEY;
output (none) RIGHT_KEY;
output (none) UP_KEY;
output (none) DOWN_KEY;

var [] byte key;
var bool press;

every(_, key, press) in CM_SCENE_KEY do
  if press then
    if key == "LEFT" then
      emit LEFT_KEY();
    else/if key == "RIGHT" then
      emit RIGHT_KEY();
    else/if key == "UP" then
      emit UP_KEY();
    else/if key == "DOWN" then
      emit DOWN_KEY();
    end
  end
end
```

Listing 5.3: A distributed version of the TV controller application. The code on the left renders the videos and the code on the right handles users input.

Up to this point we have taken the local version of this application and split it into two standalone programs that run on different devices. Note, however, that there is a key part missing: the specification of how both programs interact with each other.

In the listing above, the source code on the right emits an output event whenever an arrow key is pressed. In the source code on the left, the program reacts to four distinct input events: `PREVIOUS_VIDEO`, `NEXT_VIDEO`, `INCREASE_VOLUME` and `DECREASE_VOLUME`—we have omitted the codes that implement these actions because they are identical to their corresponding in Listing 4.12. For this example to work as expected, we have to map the remote control output events to the TV input events, as shown in Table 5.1.

In Mars, such mapping is specified through a Lua script (called *mapping script*). But before we present it, let's introduce the concept of *interface*. In a distributed Mars application, each device implements an interface that defines the set of input and output events it exposes. A Lua table (called *interface table*) defines these interfaces. The interface table is used in the pre-compilation phase and by the Mars server when it computes whether an event received

Table 5.1: Remote control output events should be mapped to the corresponding TV input event.

| **Events Mapping** | |
|---|---|
| **OUTPUT** | **INPUT** |
| RIGHT_KEY ⟶ | NEXT_VIDEO |
| LEFT_KEY ⟶ | PREVIOUS_VIDEO |
| UP_KEY ⟶ | INCREASE_VOLUME |
| DOWN_KEY ⟶ | DECREASE_VOLUME |

by a device should be forwarded to others (these topics are covered later in this chapter). Listing 5.4 depicts the interface table of the TV controller application.

```
1  --[[ hello_mars_interface.lua ]]
2  interfaces {
3    REMOTE_CONTROL = {
4      outputs = {
5        LEFT_KEY = {}, RIGHT_KEY = {}, UP_KEY = {}, DOWN_KEY = {}
6      }
7    },
8    TV = {
9      inputs = {
10       PREVIOUS_VIDEO = {}, NEXT_VIDEO = {},
11       INCREASE_VOLUME = {}, DECREASE_VOLUME = {}
12     }
13   }
14 }
```

Listing 5.4: Interface table defining two interfaces: REMOTE_CONTROL and TV.

Let's go back to the mapping script. Listing 5.5 shows a script that maps the output events of the remote control to the input events of the TV according to the mapping in Table 5.1.

```
1  --[[ hello_mars_mapping.lua ]]
2  local tv = nil
3  local control = nil
4
5  function map_events ()
6    map (control, "LEFT_KEY",  tv, "PREVIOUS_VIDEO")
7    map (control, "RIGHT_KEY", tv, "NEXT_VIDEO")
8    map (control, "UP_KEY",    tv, "INCREASE_VOLUME")
9    map (control, "DOWN_KEY",  tv, "DECREASE_VIDEO")
10 end
11
12 MARS.onConnect = function (p)
13   local interfaces = p:getInterfaces()
14
15   for i,_ in pairs (interfaces) do
16     if i == "TV" then
17       tv = p
```

```
18      elseif i == "REMOTE_CONTROL" then
19        control = p
20      end
21    end
22
23    if tv ~= nil and control ~= nil then
24      map_events()
25    end
26  end
```

Listing 5.5: Mapping script that maps remote control output events to TV input events.

The function `onConnect()` (lines 12–26) is executed by the MARS server whenever a new device connects. The script above maps all events of the remote control to the TV using the function `map()` (lines 5–10) which is part of the MARS API.

This is all we have to implement for programming the remote control and TV sample application: the CÉU source codes of the remote control and the TV, besides the interface table and the mapping script. Note that there is no need to use low-level communication primitives because the middleware handles all data exchange transparently.

Now let's change our sample distributed application to accommodate scenarios in which there are multiple remote controls and multiple TVs. The only change that we need to do is in the mapping script, because the interfaces are the same and the logic of applications also did not change. Thus, Listing 5.13 depicts a mapping script that maps output events of multiple remote controls to input events of multiple TVs. The major change is that instead of having a single variable for storing the remote control and TV instances as we have in Listing 5.5 (variables `tv` and `control`) we now have a table for storing all connected instances of those interfaces. When a new device connects, the mapping script iterates over these tables and defines the appropriate mappings.

```
1  --[[ hello_mars_mapping.lua ]]
2  local tvs = {}
3  local controls = {}
4
5  function map_events (control, tv)
6    map (control, "LEFT_KEY",  tv, "PREVIOUS_VIDEO")
7    map (control, "RIGHT_KEY", tv, "NEXT_VIDEO")
8    map (control, "UP_KEY",    tv, "INCREASE_VOLUME")
9    map (control, "DOWN_KEY",  tv, "DECREASE_VIDEO")
10 end
11
12 MARS.onConnect = function (p)
13   local interfaces = p:getInterfaces()
14   local isTv = false
15
16   for i,_ in pairs (interfaces) do
```

```
17      if i == "TV" then
18        table.insert (tvs, p)
19        isTv = true
20      elseif i == "REMOTE_CONTROL" then
21        table.insert (controls, p)
22        control = p
23      end
24    end
25
26    if isTv then
27      for _,inst in ipairs(controls) do
28        map_events (inst, p)
29      end
30    else
31      for _,inst in ipairs (tvs) do
32        map_events (p, inst)
33      end
34    end
35  end
```

Listing 5.6: Mapping script that maps multiple remote controls to multiple tvs.

At runtime, the middleware guarantees that all TVs receive the events in the same order. Next section provides more details of how a MARS distributed application is executed.

## 5.4 Executing a Mars Distributed Application

When the MARS server starts, it receives as argument a mapping script and an interface table, and it creates a session. Devices may join a session at any time by sending a join message to the server passing the interfaces they implement. The server rejects a device if it implements an unknown interface (i.e., an interface that is not specified in the interface table). We assume that the interface table is known by all joining devices.

In the Listing 5.5 and Listing 5.6, the mapping scripts implement the `onConnect()` function, which is a callback executed by the server whenever it accepts a new device in the session. There are other two callbacks the mapping script may define, which are explained ahead.

The `onConnect()` callback receives as argument an object (table) of the class `Peer` (metatable) with information about the joining device. The method `Peer.getInterfaces()`, used in the mapping scripts above, returns a list of interfaces the joining device implements.

These mapping scripts use the function `map()` to bind the output events of the remote controls to the input events of the TVs. For instance, consider the line 6 of Listing 5.6, which contains the following code: `map(control,"LEFT_KEY",tv,"PREVIOUS_VIDEO")`. This statement indicates that

the output event `LEFT_KEY` emitted by the process `control` should trigger the input event `PREVIOUS_VIDEO` on the device `tv`.

When one uses the Mars compilation process to compile a Céu source code, the Mars client runtime is attached to the final program and runs in parallel with the application. This runtime is responsible for connecting and communicating with the Mars server. It also catches all output events emitted by the program and sends them to the server. Likewise, when the server sends an input event to a device, the runtime receives this message and properly generates the corresponding Céu input event to the application.

Note that all this process is hidden from applications. For instance, the codes in Listing 5.3 have no communication primitive (e.g., send/receive) or even include any odd library. Instead, they are regular Céu codes that use the Céu-Media library, and if one compiles them using a typical Céu compilation process, the output is a valid program that runs locally with no communication with any other program (unless explicitly programmed). The compilation of a Mars application has a precompilation phase that embeds the runtime into the final binary.

## 5.5 Implementation of the timing-sequential consistency model in Mars

In the centralized Mars architecture, the server acts as moderator that defines the order of events. To illustrate, consider again the remote controls and TVs distributed application in Section 5.3. Suppose one presses the RIGHT arrow key of one control and immediately presses the LEFT arrow key of the other control.

Each output event is always sent to the Mars server in an *output message*. The server processes output messages following a FIFO (First In First Out) policy. Thus, suppose the second message, viz. the LEFT arrow key, arrives first in the server. The server will check its mapping mapping table and send an *input message* carrying the input event PREVIOUS_VIDEO to all TVs in the session. Some time later, the LEFT arrow key message arrives. Again, the server checks its mapping table and then sends the NEXT_VIDEO event to the TVs.

Because the nodes maintain a connection to the server using a protocol that guarantees message ordering (i.e., TCP) all TVs receive first the PREVIOUS_VIDEO event and then the NEXT_VIDEO. Figure 5.1 illustrates this scenario. Following this approach, the real-world time that peers send output messages is disregarded in favor of the order that messages arrive at the server.

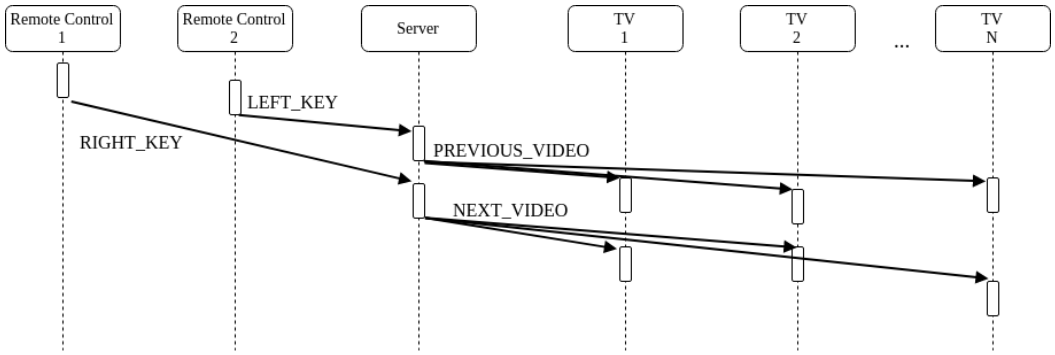


Figure 5.1: Input events are sent to peers in the same order the server processes output messages.

Note that this approach ensures the two conditions that a system should guarantee following Lamport's sequential consistency definition (page 71):

- After processing an output message, the server sends input messages in the exact same order to all processes triggered by that event, according to the current state of the mapping table—this guarantees the first condition.
- The transport protocol ensures that the server receives all messages from any process in the same order that processes send them—this guarantees the second condition.

From the distributed system point of view, this indeed guarantees consistency for applications, but this naive approach leaves room for glitches in some distributed interactive multimedia applications. For instance, consider a distributed car driving simulator operated by two users, an instructor and a student. At some moment during the simulation, an obstacle pops up and the student must dodge from it. If the student turns the steering wheel after the car exceeds the minimum safe distance from the obstacle (critical point) the car hits it. Assume that the student and the instructor are in separate rooms, and that this application runs on the following distributed setting: a student steering wheel controller device, and two simulators devices, one for the student and other for the instructor.

If one implements this application using Mars, whenever the student turns the steering wheel, an output message is sent to the server, which then sends an input message to update the simulators. However, the scenario depicted in Figure 5.2 may happen.

Note the student turns the steering wheel some time before reaching the critical point and the student simulator receives the UPDATE message on time to prevent the car to hit the obstacle. However, the instructor simulator receives

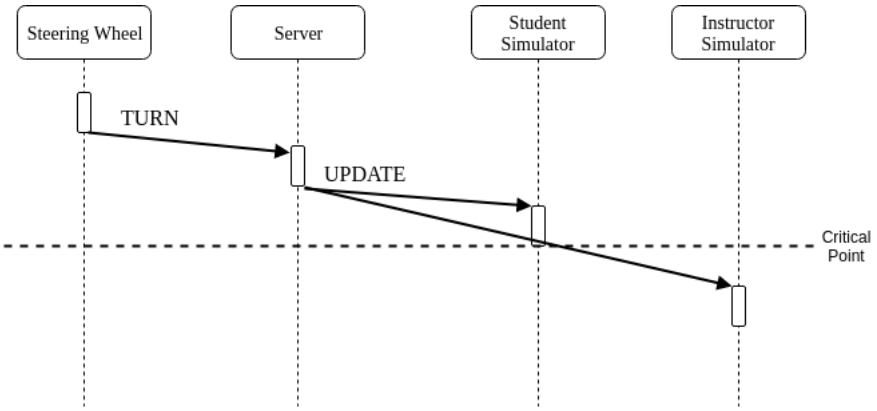


Figure 5.2: The UPDATE event reaches the instructor's simulator after the car has passed the critical point, but the student's simulator receives this message on time to dodge from the obstacle.

this message only when it is too late to avoid the car from crashing. And this may happen despite the system's guarantees regarding sequential consistency. This problem arises because in distributed interactive multimedia applications consistency is not only about ensuring total ordering of messages, but also about guaranteeing that operations are executed at the correct time [95].

### Guaranteeing timing consistency to distributed input events

Mars implements an algorithm that ensures that all Céu applications react to input events coming from the server at the same logical time.

Before the server sends input messages, it sets a timestamp to the input event. When a process receives an input message, it delays the generation of the input event until the application logical time matches with the value indicated by the event timestamp. Thus, all processes generate the input event at the same logical time, which is defined by the server.

A key point in this approach is how to define the value of this timestamp. If it is too short, input messages will likely arrive delayed at the peers, that is, the logical time of applications will be ahead of the timestamp value. If it is too long, the system responsiveness can become compromised, as well as users experience.

Our approach uses the maximum RTT (Round Trip Time) value between the Mars server and peers in the calculation of events timestamp, and the current logical time of processes in the session. Algorithm 1 depicts the pseudo-algorithm the server executes when it receives an output message.

First, the server checks the mapping table to get the list of all events that should be generated from the received output message (line 2). The return is a list of pairs $<I, P>$, in which $I$ is an input event and $P$ is a set of peers that

**Algorithm 1:** Algorithm that the MARS server runs to calculate the timestamp of input events.

```
1  foreach Output event O do
2      list<I, P> ← {check the mapping table and get triggered events
         from O};
3      T_0 ← now();
4      foreach <I_i, P_i> in list<I, P> do
5          {send a message to all peers in P_i asking for the current
           logical time };
6          T_max ← 0;
7          RTT_max ← τ;
8          N ← |P_i|;
9          while N > 0 do
10             N ← N − 1;
11             msg ← { await a response message};
12             t ← { unpack current logical time of msg };
13             if t > T_max then
14                 T_max ←t;
15             end
16             RTT ← { get RTT of msg };
17             if RTT > RTT_max then
18                 RTT_max ←RTT;
19             end
20         end
21         T_timestamp ← T_max + RTT_max + (now() − T_0) + Δ;
22         { set timestamp T_timestamp to I_i };
23         { send I_i to all peers in P_i };
24     end
25 end
```

should receive the input event $I$. At this point, the current time is saved in variable $T_0$ (line 3).

For each pair $<I_i, P_i>$, the server sends a message to all peers in $P_i$ asking for the current logical time (line 5) and then waits for the responses. Upon receiving each reply, the server updates the variables $T_{max}$, which should store the most advanced logical time reported, and $RTT_{max}$, which should store the longest RTT calculated (lines 9–20). Note that the algorithm initializes $RTT_{max}$ with the value $\tau$, which corresponds to an estimate of the mean RTT of the underlying network. This ensures a minimum value for the $RTT_{max}$ variable, whose implication is explained when we discuss the algorithm executed on the peers for ensuring the consistency.

After the server has received all responses, it can calculate the timestamp $T_{timestamp}$ of the input event $I_i$, which is the sum of the most advanced logical time reported ($T_{max}$), with the maximum RTT calculated or $\tau$, whichever is

greater ($RTT_{max}$) and with the total time elapsed waiting for the responses (now() - $T_0$) – line 21. Additionally, we add a $\Delta$ value for compensating eventual network jitter, which in our current implementation is a tenth of the maximum RTT ($\Delta = 0.1 * RTT_{max}$).

To illustrate, consider Figure 5.3. When the server receives the TURN event, it asks the current logical time of the simulators. Let's say the Student Simulator reports $4s$ and the Instructor Simulator reports $5s$. The value of the variable $T_{max}$ then is $5s$. Imagine the RTT of the message to the Student Simulator was $80ms$ and to the Instructor Simulator was $60ms$. Assuming that the value of $\tau$ is less than $80ms$, the $RTT_{max}$ then has the value $80ms$. Consider that 100ms have elapsed until the server receives all responses. In this scenario, the timestamp $T_{timestamp}$ of the event UPDATE is $5.188s$ ($T_{timestamp} = 5000 + 80 + 100 + (80 * 0.1)$).

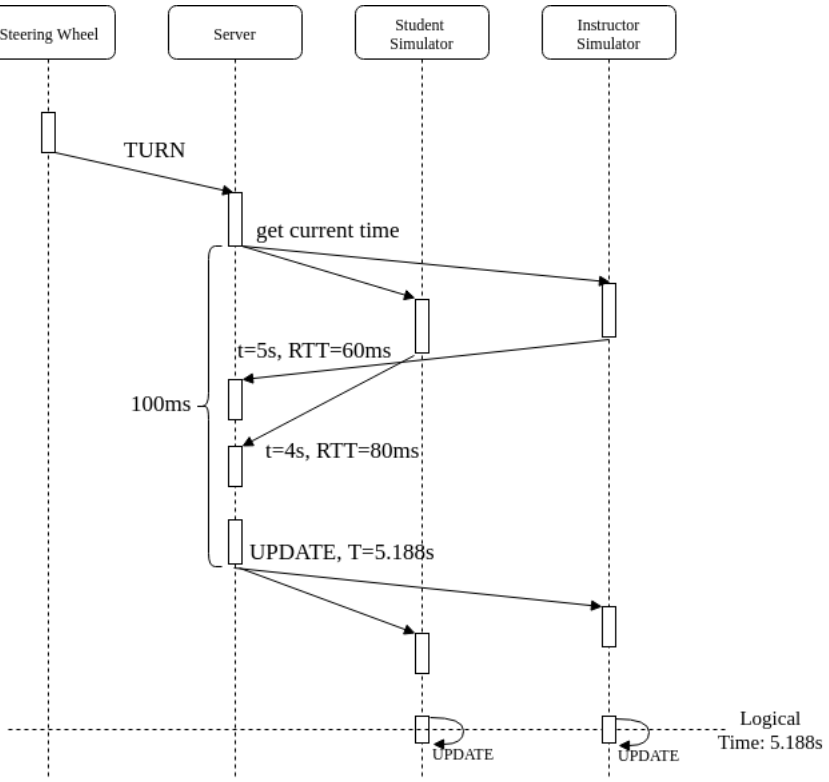


Figure 5.3: Messages exchange between the server and simulators to calculate the timestamp of the UPDATE event.

The server then sends the input messages carrying the UPDATE event with that timestamp. When the simulators receive this message, they both delay the generation of this input event until the application logical time reaches the value $5.188s$. And note that this happens even though both simulators receive the UPDATE input message at different instants.

However, occasionally the logical time of an application may be ahead of the timestamp set to the event of an input message because the server cannot

**Algorithm 2:** Algorithm that the MARS runtime runs to halt the application if it does not receive an input message before the timer expires.

```
1  foreach Message msg do
2      if msg requests the current time then
3          t ← now();
4          {send t as response};
5          spawn start_timer (τ);
6      else if msg is an input message then
7          emit cancel_timer ();
8          {Parse msg and schedule the generation of the event };
9      end
10 end
11 Function start_timer(τ):
12     par/or do
13       await τ;
14       {halt application};
15     with
16       await cancel_timer;
17     end
18     {Resume the application if it is halted};
```

always accurately predict network delay and jitter. For instance, if the time difference between the instant the Instructor Simulator receives the UPDATE message and the instant it replied back the server with its current logical time is greater than 188ms, it will not be able to generate the UPDATE event to the application precisely at the time 5.188s because this instant is in the "past".

To prevent such cases, we have developed a control mechanism that runs on peers' side. Following the simple protocol described above, whenever a peer receives a message asking for its current time, the server will subsequently send an input message. Thus, after the peer replies back the server, it starts a timer. If the timer expires before the input message arrives, the MARS runtime halts the whole application until it receives the message. In this context, by *halt* we mean the application stops to receive events, including timing. The timer is canceled if the peer receives the input message before the timer expires.

Algorithm 2 depicts the pseudocode that implements this control mechanism on clients. When a message asking for the current time arrives, the runtime replies and creates a timer using the CÉU `spawn` construct (line 5) that is, the timer starts to run immediately in parallel with this chunk of code. The function `start_timer` uses the CÉU `par/or` composition for implementing the timer: in the first trail, it waits for the time passed as argument (line 15) and halts the application when it wakes from this `await` (line 16); the second

trail just waits for the event `cancel_timer` (line 18)—emitted when the runtime receives the input message (line 7)—to then abort the timer (i.e., the `par/or` composition). After that, the runtime resumes the application if it has been halted due to the expiration of the timer.

This algorithm works as intended if the timer necessarily expires before the timestamp of the input event. For instance, consider a case that a process receives a message asking for its current time and it replies back with the value $5s$. At this point, the runtime starts a timer of, let's say, $20ms$. If the server sets the value of $T_{timestamp}$ to $5.015s$ and the message delays, when the timer expires the logical time will be ahead of the instant the event should have been generated.

In our implementation, we set the value of the timer to be the same $\tau$ value (an estimate of the mean network RTT) used by the server to initialize the variable $RTT_{max}$. With this trick, we ensure that the timer is always less than the value of $T_{timestamp}$ because the server adds other components to calculate this value (see Algorithm 1, line 21).

## 5.6 Mars Internals

This section presents the main internal components of the Mars middleware. We first discuss the server side to then present the client side.

### 5.6.1 Server Side

Figure 5.4 illustrates the main components of the Mars server. The mapping script and the interfaces table are passed as argument to the server when it starts. The other components in the figure are intrinsic parts of the server.

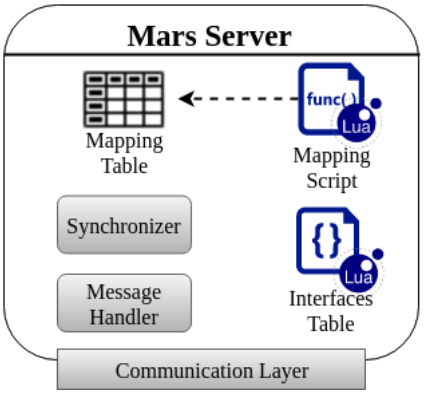


Figure 5.4: Mars server main components.

The *communication layer* deals with all low-level data exchange among the server and other devices in a session. It serializes messages to send and deserializes received messages. It is also responsible for managing messages sequencing and ordering.

The *message handler* is the logical component that interprets received messages and takes appropriate actions to handle them. If it is a *join* message, this component checks if the joining device implements a known interface. If it is an output message, the messages handler interacts with the *synchronizer*, which executes Algorithm 1.

The main goal of the mapping script is to construct the mapping table. It may define the following callbacks:

- MARS.`onConnect()`: executed when a device is accepted in the section;
- MARS.`onDisconnect()`: executed when a device disconnects;
- MARS.`onOutputEvent()`: executed when the server receives an output message.

Mapping scripts call the `map()` function (defined in the MARS API) to build the mapping table. This function has the following signature:

```
map (instanceFrom,OUTPUT_EVT,instanceTo,INPUT_EVT,filterFunc)
```

A call to the `map()` function creates an entry in the mapping table binding the `OUTPUT_EVT` from `instanceFrom` process to the `INPUT_EVT` of `instanceTo`. The last parameter is an optional filter function that defines when a mapping should be triggered. When the server receives the event `OUTPUT_EVENT` from `instanceFrom`, it calls `filterFunc` function if it has been defined. If this function returns `true`, then the server sends an input message to `instanceTo` carrying the event `INPUT_EVT`. Otherwise this mapping is ignored.

By default, `OUTPUT_EVT` arguments are passed to `INPUT_EVT`. The `filterFunc` may change this behavior by returning a table in addition to the first boolean value. The most trivial example of this feature is in the case that the types of the arguments of `OUTPUT_EVT` and `INPUT_EVT` are different. The `filterFunc` receives as input the arguments of `OUTPUT_EVT` and should return the appropriate values that will become arguments to `INPUT_EVT`. We explore this feature in some examples discussed throughout this thesis.

### 5.6.2 Client Side

The *client side* consists of codes that run on devices and communicate with the MARS server. These programs, referred to as MARS applications, are composed of two parts: the application code and the MARS runtime. The former are regular CÉU codes (i.e., the CÉU compiler accepts them *as is*) that implement the application logic. The latter is part of the MARS client middleware and takes care of all low-level communication to interact with the server. Figure 5.5 illustrates the main logical components of the MARS runtime.

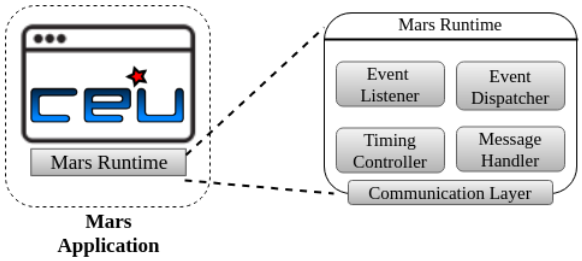


Figure 5.5: The MARS runtime runs in parallel with the application code. Its main logical components are: Events Listener, Events Dispatcher, Timing Controller and Messages Handler.

The *communication layer* is similar to the corresponding layer in the server side: it serializes, deserializes, sequences and orders messages.

The *message handler* is the component that sends and receives messages to/from the server. When it receives an input message, it activates the *timing controller* that runs Algorithm 2.

The *event dispatcher* is responsible for generating an application-level input event in response to an input message. And the *event listener* reacts to output events emitted by the application, sending them to the server.

The compilation of a MARS application has a pre-compilation phase that attaches the MARS runtime to the final program. The runtime is a non-intrusive component, i.e., it runs in parallel with the application code without interfering in its behavior. In fact, the application is neither aware that it is being executed along with the MARS runtime nor that it is part of a distributed application. Next section discusses the compilation of a MARS application.

## 5.7 Compilation

The MARS middleware is implemented using CÉU and Lua. The compilation of MARS server follows the usual process as any other CÉU program.

MARS runtime is attached to applications according to the precompilation phase illustrated in Figure 5.6.

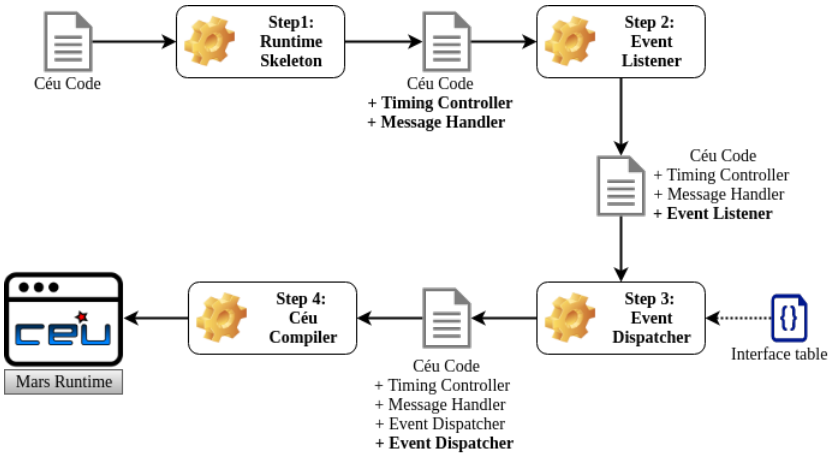


Figure 5.6: Precompilation phase of MARS applications.

The first step receives as input a CÉU source code and adds to it the components of the runtime that are independent of the application code, namely the timing controller and the message handler. If the application source code is in file *prog.ceu*, then the output of the first step is similar to the skeleton depicted in Listing 5.7. The original source code is included in the second trail of a `par/or` composition (line 22), which has the message handler and timing controller components in the first trail (lines 2–19).

```
1  par/or do
2    par/and do //messages handler
3      < send join message >
4      loop do
5        await new_message ();
6        < ... >
7        if joined then
8          emit ok;
9        end
10       < ... >
11     end
12   with      //timing controller
13     loop do
14       await time_requested ();
15       <...>
16       spawn Start_Timer ();
17       <...>
18     end
19   end
20 with
21   await ok;
22   #include "prog.ceu";
23 end
```

Listing 5.7: Output code from step 1 of precompilation phase.

Note the application does not start to run immediately when the program starts because the second trail halts waiting for the internal event `ok` (line 21).

When the runtime receives a message from the server informing that the device was accepted in the session, the message handler emits this event (line 8) which starts the application (line 22).

The second step adds the event listener component to the code. For each output event declared in the application source code, a listener is created that reacts to that event, sending it to the server. Listing 5.8 illustrates the output of the second step of the *prog.ceu* precompilation phase.

```
1  par/or do
2    par/and do
3      //messages handler
4    with
5      //timing controller
6    with
7      par do
8        output O1 do
9          < send O1 to the server >
10       end
11     with
12       output O2 do
13         < send O2 to the server >
14       end
15     with
16       <...>
17     end
18   end
19 with
20   await ok;
21   #include "prog.ceu";
22 end
```

Listing 5.8: Output code from step 2 of precompilation phase.

The last step reads the interface table and creates a native C function for each input event. These functions use the CÉU C API for generating input events to programs. Besides, this step also creates a CÉU function for handling input messages and calling the corresponding native function. The output of this last transformation is illustrated in Listing 5.9. When the logical time of the application reaches the timestamp of an internal event sent by the server, the timing controller calls the CÉU function `Emit_Input_Event()` (lines 26–34). This function checks the value passed as argument and calls the appropriate C function that generates the input event to the program (lines 1–11).

```
1  native/pre do
2    function emit_I1 (...)
3    {
4      <generate input event I1>
5    }
6    function emit_I2 (...)
7    {
8      <generate input event I2>
9    }
10   <...>
11 end
```

```
12
13 par/or do
14   par/and do
15     //messages handler
16   with
17     //timing controller
18   with
19     //events listener
20   end
21 with
22   await ok;
23   #include "prog.ceu";
24 end
25
26 function Emit_Input_Event (char[] byte evt) -> none
27 do
28   if evt == "I1" then
29     _emit_I1(...);
30   else/if evt == "I2" then
31     _emit_I2(...);
32   <...>
33   end
34 end
```

Listing 5.9: Output code from the step 3 of the precompilation phase.

The output of this last step is passed to the Céu compiler, which compiles it using the standard compilation steps. The final output is a binary having the original application and the Mars runtime attached to it.

### Quantitative analysis of the Mars Runtime impact on disk footprint

The Mars runtime has a non-negligible impact on the disk footprint of the final binary program. Table 5.2 presents the result of a quantitative comparison among programs compiled with and without the Mars runtime, regarding the size footprint. These numbers were obtained compiling the source code of nine applications in a Ubuntu 16.04 64-bits machine using Céu v0.30.

In this discussion, we call *static runtime* (SR, in the table) the output of step 1 of precompilation phase, that is, the part of the runtime that is independent of any analysis of application source code (i.e., the timing controller and message handler components). We compiled an empty Céu source code with an empty interface table using Mars compilation steps. The output was a dummy binary having just the runtime, i.e., the smallest valid Mars program possible in our current implementation. The size of this binary was $452kB$.

The first column of the table presents the size of binaries compiled using regular Céu compilation process, that is, without Mars runtime. The second column presents the size of binaries with this runtime. The third column shows

Table 5.2: Disk footprint of MARS runtime.

| | Size Footprint (kB) | | | |
|---|---|---|---|---|
| Program | $P$ | $P+R$ | $P+R-SR$ | $R(\%)$ |
| program 1 | 236 | 476 | 24 | 50.42 |
| program 2 | 228 | 480 | 28 | 52.50 |
| program 3 | 232 | 468 | 16 | 50.43 |
| program 4 | 260 | 496 | 44 | 47.58 |
| program 5 | 256 | 488 | 36 | 45.54 |
| program 6 | 244 | 488 | 36 | 50.00 |
| program 7 | 228 | 460 | 08 | 50.43 |
| program 8 | 252 | 488 | 36 | 48.36 |
| program 9 | 256 | 500 | 48 | 48.80 |

**P**: Program compiled without the MARS runtime.
**P+R**: Program compiled with the MARS runtime.
**P+R-SR**: Difference between the program compiled with the MARS runtime ($P+R$) and the static part of the runtime ($SR$).
**R(%)**: The percentage of the runtime in the binary size.

the difference between the size of final binary with the runtime minus the size of the dummy binary discussed in last paragraph. And finally, the last column presents the percentage of the MARS runtime in the final binary size.

These numbers indicate that for relative small CÉU programs, the MARS runtime has a significant impact on the final binary size. The smallest MARS program possible has almost $500kB$, which is non-negligible especially if we consider the original domain that CÉU targets, which is embedded systems. The main reason for such a footprint is that the runtime does substantial work to hide the underlying distributed infrastructure from applications. Even though we consider this footprint size acceptable, this is a point for improvement.

For completeness, it worth mentioning that the size of the MARS server is $288kb$, which is similar to the size of programs compiled without the MARS runtime depicted in the first column of Table 5.2.

## 5.8 Evaluation

We have assessed MARS performance by evaluating how the middleware behaves as the number of messages exchanged increases in a session. For this test, we have used the following setup:

– Three processes (one server and two clients) running on different machines connected via WLAN.

– All machines had the same setup: Intel Core I7 and 8Gb RAM.

One of the clients (Client 1) periodically emits an output event (called PING). The server receives these messages and generates input events (called PONG) to the other client (Client 2). Listing 5.10 illustrates the source code used in this evaluation (we have omitted the interfaces table and the mapping script in this discussion).

```
1  var s64 sum = 0;
2  var int count = 0;
3  watching 60s do
4    par do            /* Client 1 */
5      every FREQ ms do
6        var s64 payload = _now();
7        emit PING (payload);
8      end
9    with              /* Client 2 */
10     var s64 payload = await PONG;
11     every (payload) in PONG do
12       count = count + 1;
13       var s64 time_now = _now ();
14       var s64 diff = time_now - payload;
15       sum = sum + diff;
16     end
17   end
18 end
```

Listing 5.10: CÉU application used in the evaluation.

Client 1 executes the first trail of the **par** composition (lines 4–8). At each FREQ milliseconds it emits the event PING passing as payload the current wall clock (real) time. Client 2 executes the second trail (lines 9–17). It reacts to each occurrence of the event PONG, extracts the payload and calculates the difference between the time it received the event and the time it was generated. We have synchronized the clock of the machines before the tests.

Both programs run for 60s and, at the end of the execution, Client 2 calculates the mean of the time differences, that is, the mean timing offset between the generation of an event and its actual processing. We have tested this program using different values to the FREQ macro. Table 5.3 shows the events mean timing offset for different values of FREQ.

As depicted in the table, the changes in the events mean timing offset were not significant for values between 1000ms to 20ms of FREQ (we expect similar behavior for $FREQ > 1000ms$). That is, we can say that this value remained constant. However, anomalous behaviors were observed for values of FREQ below than 20ms.

Table 5.3: Mean timing offset between the generation of an event and its processing.

| FREQ | Mean timing offset (ms) |
|---|---|
| 1000 | 17.242 |
| 500 | 16.924 |
| 300 | 17.624 |
| 100 | 16.329 |
| 80 | 16.860 |
| 50 | 16.439 |
| 30 | 16.616 |
| 20 | 16.843 |

When Client 1 sends the `PING` event to the server in intervals less than 20ms, the server is not able to handle all the events: its internal buffer becomes full and some messages are dropped. This happens because the rate of input events in these cases is in the same order of magnitude than the events processing time. That is, the reaction time cannot be considered negligible, which violates the synchronous hypothesis. In other words, for these rates of input events, MARS server does not behave as a synchronous program. Similar pattern is observed in Client 2: its internal buffer becomes full and it does not receive some input messages, which results in a inconsistent value for the mean timing offset.

From this discussion, we concluded that MARS supports applications in which the rate of input events that arrives at the server is greater than 20ms.

## 5.9 Sample Applications

Here we discuss the implementation of two sample applications using MARS. The first is a modified version of the remote controls and TVs application discussed at the beginning of this chapter, and the second is a typical example of second screen applications used in the context of interactive digital TV.

### Remote Controllers and TVs (version 2)

In this version of the remote controllers and TVs, users may pause the content presented on TVs by pressing the remote control `PAUSE` button. The modified version of the remote control and TV codes are depicted in Listing 5.11.

```
input (none) PREVIOUS_VIDEO;
input (none) NEXT_VIDEO;
input (none) INCREASE_VOLUME;
input (none) DECREASE_VOLUME;

< ... >

var&? Scene s = spawn Scene (<...>);
watching (scene) do
  loop do
    var [] byte uri = <...>;

    var&? Play player =
                spawn Play (<...>);
    par/or do
      every INCREASE_VOLUME do
        < ... >
      end
    with
      every DECREASE_VOLUME do
        < ... >
      end
    with
      await NEXT_VIDEO;
      < ... >
    with
      await PREVIOUS_VIDEO;
      < ... >
    with
      await CM_PLAYER_STOP;
    with
      var bool isPaused = false;
      every PAUSE do
        if not isPaused then
          call Scene_Pause (s);
          isPaused = true;
        else
          call Scene_Resume (s);
          isPaused = false;
        end
      end
    end
  end
end
```

```
output (int) KEY_PRESSED;

var [] byte key;
var bool press;

every(_, key, press) in CM_SCENE_KEY do
  if press then
    if key == "LEFT" then
      emit KEY_PRESSED(1);
    else/if key == "RIGHT" then
      emit KEY_PRESSED(2);
    else/if key == "UP" then
      emit KEY_PRESSED(3);
    else/if key == "DOWN" then
      emit KEY_PRESSED(4);
    else/if key == "PAUSE" then
      emit KEY_PRESSED(5);
    end
  end
end
```

Listing 5.11: An alternative version of TV controller application. In this version, users may pause the video on TV by pressing the PAUSE button on the remote control.

The main difference regarding the TV source code (on the left) and the previous version is the last trail of the `par/or` composition (lines 31–42) that waits for the event `PAUSE` and then either pauses or resumes the `Scene` depending on its previous state. The remote control code (on the right) has major changes. Instead of emitting a different output event to each button, it emits the same event (`KEY_PRESSED`) passing a different code (argument) to identify the pressed button. Therefore, this requires a change in the interface table, as well as in the mapping script.

Listing 5.12 depicts the modified interface table. Note the `REMOTE_CONTROL` interface has only one output event (`KEY_PRESSED`) with an `int` parameter. The `TV` interface is similar to the original example, but it has an additional `PAUSE`

input event, with no parameters.

```
1  --[[ hello_mars_interface_v2.lua ]]
2  interfaces {
3    REMOTE_CONTROL = {
4      outputs = {
5        KEY_PRESSED = {"int"}
6      }
7    },
8    TV = {
9      inputs = {
10       PREVIOUS_VIDEO = {}, NEXT_VIDEO = {},
11       INCREASE_VOLUME = {}, DECREASE_VOLUME = {},
12       PAUSE = {}
13     }
14   }
15 }
```

Listing 5.12: A modified version of the interface table.

The modified mapping script is depicted in Listing 5.13. This script is similar to the previous version, but it uses filter functions when defining the events mapping. Because the remote control interface only defines the `KEY_PRESSED` output event, it is used in all mappings (lines 6–15) and filter functions define when these mappings are valid.

Consider the user has pressed the `LEFT` button. In this case, the argument of the `KEY_PRESSED` event is 1 (see Listing 5.11). Therefore, only the mapping in lines 6-7 is triggered, because it is the only one for which the filter function returns **`true`** as the first argument. Because the `PREVIOUS_VIDEO` has no arguments, the filter function returns **`nil`** as second value (remember from Section 5.6 that the second value returned becomes the arguments for the input event). The same reasoning applies to other output events emitted by the remote control.

```
1  --[[ hello_mars_mapping_v2.lua ]]
2  local tvs = {}
3  local controls = {}
4
5  function map_events (control, tv)
6    map (control, "KEY_PRESSED", tv, "PREVIOUS_VIDEO",
7      function (from, to, key) return key == 1, nil end)
8    map (control, "KEY_PRESSED", tv, "NEXT_VIDEO"
9      function (from, to, key) return key == 2, nil end)
10   map (control, "KEY_PRESSED", tv, "INCREASE_VOLUME"
11     function (from, to, key) return key == 3, nil end)
12   map (control, "KEY_PRESSED", tv, "DECREASE_VIDEO"
13     function (from, to, key) return key == 4, nil end)
14   map (control, "KEY_PRESSED", tv, "PAUSE"
15     function (from, to, key) return key == 5, nil end)
16 end
17
18 MARS.onConnect = function (p)
19   local interfaces = p:getInterfaces()
20
21   for i,_ in pairs (interfaces) do
```

```
22      if i == "TV" then
23        table.insert (tvs, p)
24        isTv = true
25      elseif i == "REMOTE_CONTROL" then
26        table.insert (controls, p)
27        control = p
28      end
29    end
30
31    if isTv then
32      for _,inst in ipairs(controls) do
33        map_events (inst, p)
34      end
35    else
36      for _,inst in ipairs (tvs) do
37        map_events (p, inst)
38      end
39    end
40  end
```

Listing 5.13: A modified version of the mapping script.

Our approach for maintaining the consistency of the system ensures that all TVs react at the same logical time to input events generated. In this application, a practical implication is that when one presses the `PAUSE` button, Mars guarantees that all videos pause at the exact same frame, as illustrated by the screenshot of this application in Figure 5.7.


Figure 5.7: Mars guarantees that all TVs always pause on the same frame.

### Second Screen

The application discussed in this section is a typical example of second screen applications used in the context of interactive digital TV. A content is presented on TV, and at a given moment the user is asked to choose on a personal device which content s/he wants to watch. Depending on the choice, the application shows a different content on TV. The source code of

the program running on the TV and on the personal device is depicted in Listing 5.14.

```
1  input (int) USER_CHOICE;
2  output (none) INTERACTIVITY;
3
4  var[] byte video = "vid.mp4";
5  var Properties.Video prop = val
6    Properties.Video (<...>);
7
8  var&? Scene s = spawn Scene (<...>);
9  watching s
10 do
11   var int choice = 1;
12   par/or do
13     await Play (&video, &prop, &s);
14   with
15     await 30s;
16     emit INTERACTIVITY ();
17     choice = await USER_CHOICE;
18   end
19   var[] byte path;
20   if choice == 1 then
21     path = [] .. "path1.mp4";
22   else
23     path = [] .. "path2.mp4";
24   end
25   prop = val
26     Properties.Video (<...>);
27   await Play (&path, &prop, &s);
28 end
```

```
1  input (none) SHOW_OPTIONS;
2  output (int) FINAL;
3
4  var&? Scene s = spawn Scene (<...>);
5  watching s
6  do
7    var[] byte op1 = [] .. "op1.png";
8    var[] byte op2 = [] .. "op2.png";
9
10   var Properties.Image prop1 = val
11       Properties.Image (<...>);
12   var Properties.Image prop2 = val
13       Properties.Image (<...>);
14
15   await SHOW_OPTIONS;
16   var Play p1;
17   var Play p2;
18   par/or do
19     par do
20       p1= spawn Play(&op1,&prop1 &s);
21     with
22       p2= spawn Play(&op2,&prop1,&s);
23     end
24   with
25     var uint obj;
26     obj= await CM_PLAYER_MOUSE_CLICK;
27     if obj == p1 then
28       emit FINAL (1);
29     else
30       emit FINAL (2);
31     end
32   end
33 end
```

Listing 5.14: A typical example of second screen application. The code on the left runs on the TV and the code on the right runs on a personal device.

Let's first discuss the code that runs on TV. As usual, it spawns a CÉU-MEDIA `Scene` (line 8) and creates a **`par/or`** composition (lines 12–28). First trail plays the main video (line 13) and second trail waits $30s$, emits the output event `INTERACTIVITY`, and then waits the input event `USER_CHOICE`.

The **`par/or`** composition may end either when the second trail wakes from this **`await`** (i.e., the user has chosen a content to watch) or when the main video ends (i.e., first trail ends). If the user has interacted on his/her personal device, the chosen content will be presented, otherwise the default content is selected (lines 20–27).

The code that runs on secondary device also starts spawning a `Scene` (line 4). The program halts waiting for the input event `SHOW_OPTIONS` in line 15. When it wakes from this **`await`**, two options are displayed in parallel (lines 19-23) and the program waits for the user to select one of them (line 26). Depending on the user choice, the program emits the output event `FINAL` passing the proper argument (lines 27–31).

The interface table for this application is depicted in Listing 5.15. It defines two interfaces, namely `MAIN` and `SECONDARY`. The former has one input (`USER_CHOICE`, with an `int` argument) and one output (`INTERACTIVITY`, with no argument) event. Similarly, the latter also has one input (`SHOW_OPTIONS`, with no argument) and one output (`FINAL`, with an `int`) event.

```
1  --[[ second_screen_interface.lua  ]]
2  interfaces {
3    MAIN = {
4      inputs = {
5        USER_CHOICE = {"int"}
6      },
7      outputs = {
8        INTERACTIVITY = {},
9      }
10   },
11   SECONDARY = {
12     inputs = {
13       SHOW_OPTIONS = {}
14     },
15     outputs = {
16       FINAL = {"int"},
17     }
18   }
19 }
```

Listing 5.15: The interface table of second screen application.

Listing 5.16 shows the mapping script of this application. It also implements the `onConnect` callback and suits scenarios in which there are multiple `MAIN` and/or `SECONDARY` devices, which are saved in tables `main` and `sec`, respectively (lines 1–2). When a device joins the session, the script adds it to the appropriate table, depending on the interfaces it implements (lines 5–17). The script then loops through the `main` and `sec` tables for mapping events of the already joined devices to the joining instance. This mapping happens as follows: when a `MAIN` device emits output event `INTERACTIVITY`, in `SECONDARY` devices the input event `SHOW_OPTIONS` should be generated; likewise, output event `FINAL` emitted by a `SECONDARY` device should trigger input event `USER_CHOICE` in `MAIN` devices (lines 19–31).

```
1  local main = {}
2  local sec = {}
3
4  MARS.onConnect = function (p)
5    local interfaces = p:getInterfaces()
6    local isMain = false
7    local isSec = false
8
9    for i in pairs (interfaces) do
10     if i == "MAIN" then
11       isMain = true
12       table.insert (main, p)
13     elseif i == "SECONDARY" then
14       isSec = true
```

```
15        table.insert (sec, p)
16      end
17    end
18
19    if isMain then
20      for _, sec in ipairs (sec) do
21        map (p,    "INTERACTIVITY", sec, "SHOW_OPTIONS")
22        map (sec, "FINAL",          p,    "USER_CHOICE")
23      end
24    end
25
26    if isSec then
27      for _, main in ipairs (main) do
28        map (main, "INTERACTIVITY", p,    "SHOW_OPTIONS")
29        map (p,    "FINAL",         main, "USER_CHOICE")
30      end
31    end
32  end
```

Listing 5.16: The mapping script of the second screen application.

Note that the behavior of this application may be used for implementing several different second screen applications, with minor tweaks. And, again, there is no need for users to program the communication among devices, because Mars middleware takes care of all data exchange following the mapping table built from the mapping script.

## 5.10 Discussion

One of the conclusions of the work described in this chapter is that it is possible to use the GALS approach for guaranteeing the timing-sequential consistency model to applications. The development of Mars is an indication that this is possible using a centralized architecture.

Our investigation has also pointed out that guaranteeing sequential consistency is not enough for distributed interactive multimedia applications. Besides consistency, timing guarantees should be implemented, otherwise applications can reach consistent but incorrect states. Again, Mars is an evidence that the GALS approach is suitable for providing such guarantees. In fact, our approach strongly relies on the locally synchronous behavior of process to define algorithms with a precise control over the application time.

A drawback of our approach is that it does not scale well. It has a single point of failure (the Mars server) and the entire execution of the system depends on it. This middleware has not been designed for supporting massive distributed applications, with hundreds or thousands of nodes exhaustively emitting output events, which breaks the synchronous hypothesis on local processes as discussed in the next chapter. But rather, it was designed for small to medium distributed applications (few dozen devices) running on a

local network and exchanging data. Even though failure is an important topic of any distributed system, it is out of the scope of this thesis and is left for future works.

Finally, but not least, MARS promotes the decoupling of the application logic and communication among devices. One of our main goals when designing MARS was to support the development of distributed interactive multimedia applications without programmers have to worry about implementing the low-level communication layer. As discussed in Chapter 1, the distributed systems and multimedia research communities have been doing important advances in their fields, and this work intends to contribute for filling the gap among these both research areas.

In the next chapter we discuss how real-world distributed interactive multimedia applications proposed by the research community may be implemented using MARS.

# 6
# Distributed Interactive Multimedia Applications: Using Mars in Real-World Examples

In this chapter we describe how real-world distributed interactive multimedia applications may be implemented using Mars. These use cases were proposed by researchers from the multimedia field, being one of them discussed in a research paper [94] and the others proposed by W3C Groups [96, 97].

We have chosen these use cases because they cover different aspects of multi-device applications. The first use case requires that both devices process events at the same time so they can pause the video in the same frame. The second use case requires that all students receive messages from the teacher in order so they can properly update their states and be notified about who has the control over the video. The third use case requires that both players always execute the events in the same order and at the same time, otherwise the playability of the game may be compromised. And the fourth use case requires distributed synchronization.

Here we point out which features of Mars are suitable for implementing part of these use cases, but we also discuss some limitations of our approach.

## 6.1
## Use Case 1: Social Viewing and Media Control

Alice and Bob would like to watch a video episode together, even if Alice is on a train and Bob stays at home. If Alice pauses the video while briefly speaking with the conductor, Bob's video pauses too. Alice and Bob may always trust the other to see the exact same thing, making it very easy for them to maintain a conversation, for instance by using a chat service or the phone. It would also be possible for Alice and Bob to split temporarily, that is, each having his/her own experience without synchronization points (adapted from the W3C Web and TV Interest Group [97]).

Listing 6.1 depicts a chunk of Céu code that implements this application. It has one input event (`TOGGLE_VIDEO_STATE`, line 1) and two output events (`TOGGLE_TOGETHERNESS` and `SPACE`, lines 2–3). The application spawns a Céu-Media scene (line 9) and starts to play a given video (line 11). From this point on, the program executes two trails in parallel (lines 12–31). The first

reacts to each occurrence of the event `TOGGLE_VIDEO_STATE` and pauses or resumes the scene depending on its current state (lines 15–22). The second reacts to CÉU-MEDIA key input events (lines 23–30). If the key is `"space"`, it emits the event `SPACE`, or if the key is `"escape"`, it emits the event `TOGGLE_TOGETHERNESS`.

```
1  input (none) TOGGLE_VIDEO_STATE;
2  output (none) TOGGLE_TOGETHERNESS;
3  output (none) SPACE;
4
5  var[] byte video = <video>;
6
7  var Properties.Video p = val Properties.Video (<...>);
8
9  var&? Scene scene = spawn Scene(<...>);
10 watching scene do
11   spawn Play (&video, &p, &scene);
12   par do
13     var bool isPaused = true;
14     every TOOGLE_VIDEO_STATE do
15       if isPaused then
16         call Scene_Resume (scene);
17       else
18         call Scene_Pause (scene);
19       end
20       isPaused = not isPaused;
21     end
22   with
23     var char key = _;
24     every (key) in CM_SCENE_KEY do
25       if key == "space" then
26         emit SPACE();
27       else/if key == "escape" then
28         emit TOGGLE_TOGETHERNESS ();
29       end
30     end
31   end
32 end
```

Listing 6.1: CÉU application that implements the first use case.

Listing 6.2 defines the interface table of this application. It has only one interface (`USER`) which has one input event and two output events.

```
1  interfaces {
2    USER = {
3      inputs = {
4        TOGGLE_VIDEO_STATE = {}
5      },
6      outputs = {
7        TOGGLE_TOGETHERNESS = {}, SPACE = {}
8      },
9    }
10 }
```

Listing 6.2: Interface table of the first use case.

The mapping script is depicted in Listing 6.3. It stores peer instances in the table `peers` (line 1). The variable `together` controls whether peers should pause together or not. The function `handler` is used as filter function (lines 4–

6). This script implements two MARS callbacks: `onConnect` (lines 8–16) and `onOutputEvent` (lines 18–22). In the `onConnect` callback, the script connects the output event `SPACE` to the input event `TOGGLE_VIDEO_STATE` of the joining instance. The `onConnect` callback then iterates over the `peers` table and properly connects the output event of the joining device to the input event of other peers, and vice-versa. The `onOutputEvent` toggles the boolean value stored in the variable `together` whenever any peer emits the output event `TOGGLE_TOGETHERNESS`.

```
1  local peers = {}
2  local together = true
3
4  function handler (from, to, args)
5    return together, nil
6  end
7
8  MARS.onConnect = function (p)
9    map (p, "SPACE", p, "TOGGLE_VIDEO_STATE")
10
11   for _,instance in ipairs(peers) do
12     map (p,        "SPACE", instance, "TOGGLE_VIDEO_STATE", handler)
13     map (instance, "SPACE", p,        "TOGGLE_VIDEO_STATE", handler)
14   end
15   table.insert(peers, p)
16 end
17
18 MARS.onOutputEvent = function (from, evt, args)
19   if (evt == "TOGGLE_TOGETHERNESS") then
20     together = not together
21   end
22 end
```

Listing 6.3: Mapping script of the first use case.

When any peer emits the output event `SPACE`, the input event `TOGGLE_VIDEO_STATE` is generated for that peer, according to the mapping in line 9. However, due to the mappings in lines 12 and 13, the server should also generate the `TOGGLE_VIDEO_STATE` to all peers for which the `handler` function returns `true`. This function simply returns the current value of the variable `together`. Therefore, when any peer emits the event `SPACE`, all peers receive the input event `TOGGLE_VIDEO_STATE` if `together == true` holds, otherwise only the peer that has emitted that event receives the corresponding input.

In sum, if the *togetherness* is enabled, all devices pause together at the same frame when one presses the key `"space"`. If not, only the device that has generated the `SPACE` event pauses (i.e., devices split temporally, as described in the use case).

## 6.2 Use Case 2: Online Education

Alice teaches an online course for international students from across the globe. At the time of the class, she requests the video and the first slide to be presented. Slides are presented, on her view and on all the views of her connected students, at exactly the same time. At some point one of the students has a question related to the movie. Alice temporarily gives the student access to control the video, and the student rewinds it to explain the origin of his question. Afterwards, Alice withdraws the controls from the student and continues (adapted from the Timing Object W3C draft spec [96]).

Our implementation of this use case is composed of two programs: one that runs on Alice's device (teacher program) and another that runs on students' devices (student program). Listing 6.4 depicts the source code of the former. It has three output events: `NEW_SLIDE`, `REPLY_CONTROL` and `PERFORM_SEEK`; and four input events: `REQUEST_CONTROL`, `SEEK_REQUEST`, `SHOW_SLIDE` and `SEEK`. The program emits the `NEW_SLIDE` event when a new slide should be presented on all devices (Alice's included). The `REPLY_CONTROL` event is emitted when Alice gives/withdraws the control over the video to/from some student. And the `PERFORM_SEEK` event is emitted to indicate to all devices that a seek operation should be executed. When the program receives the event `REQUEST_CONTROL`, it means that a student has requested control over the video, and the control should be granted if there is no other student currently controlling it. When a student wants to rewind the video, the program receives the event `SEEK_REQUEST`. After receiving the event `SHOW_SLIDE`, The teacher program should exhibit the slide indicated in the argument of the event. And the `SEEK` event should trigger a seek operation.

```
1  output (int) NEW_SLIDE;
2  output (int) REPLY_CONTROL;
3  output (int) PERFORM_SEEK;
4
5  input (int) REQUEST_CONTROL;
6  input (int) SEEK_REQUEST;
7  input (char) SHOW_SLIDE;
8  input (int) SEEK;
9
10 <...>
11
12 var&? Scene scene = spawn Scene(<...>);
13 var int control;
14 watching scene do
15   var&? Play player = spawn Play (&video, &prop, &scene);
16   emit NEW_SLIDE (0);
17   par do
18     var int slide = 0;
19     var char key = _;
20     every (key) in CM_SCENE_KEY do
```

```
21        if key == "space" == 0 then
22          slide = slide + 1;
23          emit NEW_SLIDE (slide);
24        end
25      end
26    with
27      var int id = 0;
28      every (id) in REQUEST_CONTROL do
29        if control == 0 then
30          control = id;
31        end
32        emit REPLY_CONTROL (control);
33      end
34    with
35      var int position = _;
36      every (position) in SEEK_REQUEST do
37        emit PERFORM_SEEK (position);
38      end
39    with
40      var char path = _;
41      every path in SHOW_SLIDE do
42        spawn Slide_Player (&path, &scene);
43      end
44    with
45      var int position = _;
46      every (position) in SEEK do
47        call Player_Seek (&player, position);
48        control = 0;
49        emit REPLY_CONTROL (control);
50      end
51    end
52  end
```

Listing 6.4: CÉU source code of the teacher program.

After creating a `Scene` (line 12) the program starts the video and emits the `NEW_SLIDE` event passing 0 as argument, which indicates that the first slide should be presented (lines 15–16). The program then creates a parallel composition with 5 trails (lines 17–51). The fist requests a new slide to be presented whenever Alice presses the `"space"` key (lines 18–25). The second waits for event `REQUEST_CONTROL` and, if there is no student controlling the video (i.e., the variable `control` is 0) it broadcasts to all students that one of them is now controlling it (i.e., it emits the event `REPLY_CONTROL` passing a student identifier)—lines 27–33. The third just waits for event `SEEK_REQUEST` and emits event `PERFORM_SEEK` (lines 35–38). The fourth changes the current slide in response to the event `SHOW_SLIDE` (lines 40–43). And finally, the last trail reacts to each occurrence of the event `SEEK`, performing a seek operation, and then withdraws the control from the student (i.e., it sets the variable `control` to 0) broadcasting this to all students (lines 45–50).

Now let's focus on the students program, whose source code is depicted in Listing 6.5. It has two output events: `TRY_GET_CONTROL` and `REWIND`; and

three input events: `SHOW_SLIDE`, `SEEK` and `CONTROL_CHANGED`. When a student wants to ask a question, it emits the event `TRY_GET_CONTROL`, and the event `REWIND` to actually request a rewind operation on the video. The input events `SHOW_SLIDE` and `SEEK` have similar purposes than their corresponding in the teacher program. The input event `CONTROL_CHANGED` is received when the student controlling the video changes.

```
1  output (int) TRY_GET_CONTROL;
2  output (int, int) REWIND;
3
4  input (char) SHOW_SLIDE;
5  input (int) SEEK;
6  input (int) CONTROL_CHANGED;
7
8  var int self = < student ID>;
9
10 <...>
11
12 watching scene do
13   var&? Play player = spawn Play (&video, &prop, &scene);
14   par do
15     var char path = _;
16     every path in SHOW_SLIDE do
17       spawn Slide_Player (&path, &scene);
18     end
19   with
20     var char key = _;
21     every (key) in CM_SCENE_KEY do
22       if key == "Return" then
23         emit TRY_GET_CONTROL (self);
24       else/if key == "space" then
25         var int position = < get rewind video position >;
26         emit REWIND (self, position);
27       end
28     end
29   with
30     every (id) in CONTROL_CHANGED do
31       if id == self then
32         <this student is now controlling the video>
33       else
34         <someone else is controlling the video>
35       end
36     end
37   with
38     var int position = _;
39     every (position) in SEEK do
40       call Player_Seek (&player, position);
41     end
42   end
43 end
```

Listing 6.5: Céu source code of the students program.

Each student has an id, which is stored in the variable `self` (line 8). After creating a `Scene`, the program starts to render the video of the class (line 13). It then creates a parallel composition with four trails (lines–14–42). The first

reacts to the input event `SHOW_SLIDE` and changes the current slide being presented (lines 15–18). The second reacts to key events as follows: if the key pressed is `"Return"`, then the program emits the `TRY_GET_CONTROL` event; otherwise, if it is `"space"`, the program gets the position to which the student wants to rewind the video and then emits the `REWIND` event passing the student identifier and the position (lines 20–28). The third waits for the event `CONTROL_CHANGED`: if the argument is equals the variable `self`, then this student has the control—at this point, the program could show some message on the screen to inform the student that s/he can seek the video (lines 30–36). And the last trail reacts to `SEEK` events, updating the current position of the video (lines 38–42).

The interface table of this use case is presented in Listing 6.6. There are two interfaces: `TEACHER` and `STUDENT`, each with its own set of input and output events.

```
1  interfaces {
2    TEACHER = {
3      inputs = {
4        SHOW_SLIDE_ = {"char"}, SEEK = {"int"},
5        REQUEST_CONTROL = {"int"}, SEEK_REQUEST = {"int"}
6      },
7      outputs = {
8        NEW_SLIDE = {"int"}, REPLY_CONTROL = {"int"}, PERFORM_SEEK = {"int"}
9      },
10   },
11   STUDENT = {
12     inputs = {
13       SHOW_SLIDE = {"char"}, SEEK = {"int"}, CONTROL_CHANGED = {"int"}
14     },
15     outputs = {
16       TRY_GET_CONTROL = {"int"}, REWIND = {"int", "int"}
17     }
18   }
19 }
```

Listing 6.6: Interface table of the online education use case.

In this example, the mapping script is a bit more complex than the others discussed so far. Listing 6.7 presents its code.

```
1  local teacher = nil
2  local students = {}
3
4  MARS.onConnect = function (p)
5    local interfaces = p:getInterfaces ()
6
7    for i,_ in pairs (interfaces) do
8      if i == "TEACHER" then
9        teacher = p
10       map (p, "NEW_SLIDE", p, "SHOW_SLIDE", slideTransf)
11       map (p, "PERFORM_SEEK", p, "SEEK")
12
13       for _, s in pairs (students) do
14         link (teacher, s)
```

```
15        end
16      elseif i == "STUDENT" then
17        table.insert (students, p)
18        if teacher ~= nil then
19          link (teacher, p)
20        end
21      end
22    end
23  end
24
25  local hasControl = nil
26  MARS.onOutputEvent = function (from, evt, args)
27    if (from == teacher and evt == "REPLY_CONTROL") then
28      if (args[1] == 0) then
29        hasControl = nil
30      else
31        hasControl = args[1]
32      end
33    end
34  end
35
36  function link (teacher, student)
37    map (teacher, "NEW_SLIDE", student, "SHOW_SLIDE", slideTransf)
38    map (teacher, "REPLY_CONTROL", student, "CONTROL_CHANGED")
39    map (teacher, "PERFORM_SEEK", student, "SEEK")
40    map (student, "TRY_GET_CONTROL", teacher, "REQUEST_CONTROL")
41    map (student, "REWIND", teacher, "SEEK_REQUEST",
42      function (from, to, args)
43        if (from == hasControl) then
44          return true, {args[2]}
45        else
46          return false, nil
47        end
48      end)
49  end
50
51  function slideTransf (from, to, args)
52    return true, {"slide" .. args[1] .. ".jpg"}
53  end
```

Listing 6.7: Mapping script of the online education use case.

The mapping script implements two callbacks: `onConnect` (lines 4–23) and `onOuputEvent` (lines 26–34). It assumes that there is only one teacher (variable `teacher`, line 1) and many students (table `students`, line 2). The `onConnect` callback gets a list of interfaces each joining peer implements. If it implements the TEACHER, then the script saves this instance in the variable `teacher` and maps its output events `"NEW_SLIDE"` and `"PERFORM_SEEK"` to its input events `"SHOW_SLIDE"` and `"SEEK"`, respectively.

Note, however, that the events `"NEW_SLIDE"` and `"SHOW_SLIDE"` are incompatible because their arguments have different types: the former event has a **`int`** argument, and the latter has a **`char`** argument. Thus, we use the filter function `slideTransf` (lines 51–53) as follows: this function always returns **`true`** as first value because we always want to trigger the mapping; the second argument

creates a table having just a `string` (the argument of the `"SHOW_SLIDE"` event) which converts the value of the event `"NEW_SLIDE"` into a string having the path of the slide to be presented.

Back to the mapping script, it iterates over the `student` table in lines 13–15 and 18–20, calling the function `link` (lines 36–49) to define the other mappings of this application. These maps are:

- teacher's `"NEW_SLIDE"` event to students' `"SHOW_SLIDE"` event, also using the `slideTransf` function;
- teacher's `"REPLY_CONTROL"` event to students' `"CONTROL_CHANGED"`;
- teacher's `"PERFORM_SEEK"` event to students' `"SEEK"`;
- students' `"TRY_GET_CONTROL"` event to teacher's `"REQUEST_CONTROL"`;
- students' `"REWIND"` event to teacher's `"SEEK_REQUEST"`, using a filter function that tests if the student who has emitted the `"REWIND"` indeed is the one who has the control over the video.

And in the callback `onOutputEvent`, the mapping script updates the value of the variable `hasControl`, which stores the identifier of the student who is controlling the video.

## 6.3 Use Case 3: Multiplayer Shooting Game

Alice and Bob like to play a multiplayer shooting game. Each player controls his/her own avatar and plays against the other. The avatars are in a virtual world having obstacles and hiding spots. The goal of each player is to find the opponent's avatar and shoot him. In Listing 6.8 there is a Céu code that implements the main functionalities of this game (adapted from a distributed consistency problem discussed by Mauve et al. [94]).

```
1  input (int, int) OPPONENT_POS;
2  input (int, int) OPPONENT_MOVE;
3  input (int, int) MY_MOVE;
4  input (bool, int, int) SHOT;
5
6  output (int, int) MY_POS;
7  output (int, int) UPDATE_POS;
8  output (int, int) FIRE;
9
10 event (none) update;
11
12 var Position self;
13 var Position opponent;
14
15 par/and do
16   self = < get initial position () >;
17   emit MY_POS (self.x, self.y);
```

```
18 with
19   var int x;
20   var int y;
21   (x, y) = await OPPONENT_POS ();
22   opponent.x = x;
23   opponent.y = y;
24 end
25
26 var&? Scene scene = spawn Scene(<...>);
27 watching scene
28 do
29   par do
30     <draw virtual world>
31   with
32     var char key;
33     var int delta = 10;
34     every key in CM_SCENE_KEY do
35       if key == "Left" then
36         emit UPDATE_POS (self.x - delta, self.y);
37       else/if key == "Right" then
38         emit UPDATE_POS (self.x + delta, self.y);
39       else/if key == "Up" then
40         emit UPDATE_POS (self.x, self.y - delta);
41       else/if key == "Down" then
42         emit UPDATE_POS (self.x, self.y + delta);
43       else/if key == "space" then
44         var int x;
45         var int y;
46         (x, y) = <get shoot direction>;
47         emit FIRE (x, y);
48       end
49     end
50   with
51     var int x;
52     var int y;
53     every (x, y) in MY_MOVE do
54       self.x = x;
55       self.y = y;
56       emit update;
57     end
58   with
59     var int x;
60     var int y;
61     every (x, y) in OPPONENT_MOVE do
62       opponent.x = x;
63       opponent.y = y;
64       emit update;
65     end
66   with
67     every update do
68       <update the position of avatars>
69     end
70   with
71     var int is_my_shot;
72     var int x;
73     var int y;
74     every (is_my_shot, x, y) in SHOT do
75       if is_my_shot then
76         <check if opponent was hit>
```

```
77        else
78          <check if my avatar was hit>
79        end
80      end
81    end
82  end
```

Listing 6.8: CÉU source code of the multiplayer shooting game.

For simplicity, this code assumes a two-player game, each runnning this program in his/hers own device. The program has four input and three outputs events (lines 1–8). The event `OPPONENT_POS` informs the program the initial position of the opponent. Whenever the opponent or the player makes a movement, the program receives the events `OPPONENT_MOVE` or `MY_MOVE` to update the game state, respectively. And when one of the players shoots the other, the event `SHOT` is received.

At the beginning of the game, the program assigns a random position to each player. The game emits the event `MY_POS` to update the other player about this initial position. When players want to change the position of their avatar, they should use one of the arrow keys, which makes the program emit event `UPDATE_POS`. And when a player wants to shoot the other, event `FIRE` is emitted.

The program begins creating a **`par/and`** composition that calculates a random position and emits event `MY_POS` in the first trail, and waits for opponent position in the second trail (lines 15–24). The player and opponent positions are stored in the variables `self` and `opponent`, respectively.

After this initial setup, the program creates a `Scene` and the game actually begins (lines 26–82). The whole logic of the game is implemented in the **`par`** composition in lines 29–81. In the first trail, the program draws the virtual world interface (omitted in this code for simplicity). The second trail reacts to input key events as follows: arrow keys lead the program to emit the event `UPDATE_POS`, passing the updated position as argument; and the space key makes the program to emit the event `FIRE`, passing the direction of the shot as argument.

The third and fourth trails are similar: they react to the event `MY_MOVE` or `OPPONENT_MOVE` to update the variables `self` and `opponent`, and then emit the internal event `update`. The fifth trail reacts to each occurrence of the event `update` to refresh the avatars' position according to the values in those two variables.

The last trail reacts to the event `SHOT`, whose first parameter indicates whether the shot was fired by the player (`is_my_shot == true`) or not (`is_my_shot == false`). Depending on this first value, the program checks if the shot in direction `x,y` hit the opponent or the player's avatar.

This application has just one interface (`PLAYER`), as shown in Listing 6.9. The events exposed in the interface table are the same discussed above: `OPPONENT_POS`, `OPPONENT_MOVE`, `MY_MOVE`, `SHOT`, `MY_POS`, `UPDATE_POS` and `FIRE`.

```
interfaces {
  PLAYER = {
    inputs = {
      OPPONENT_POS = {"int", "int"}, OPPONENT_MOVE = {"int", "int"},
      MY_MOVE = {"int", "int"}, SHOT = {"bool", "int", "int"}
    },
    outputs = {
      UPDATE_POS = {"int", "int"}, MY_POS = {"int", "int"},
      FIRE = {"int", "int"}
    }
  }
}
```

Listing 6.9: Interface table of the game.

Now let's discuss the mapping script in Listing 6.10. This script defines only the `onConnect` callback. The event `MY_POS` of each player is mapped to the event `OPPONENT_POS` of the other. The event `UPDATE_POS` of each player has two mappings: one to the event `OPPONENT_MOVE` in the opponent device (lines 13–14), and another to the event `MY_MOVE` in the same device (line 20). That is, when this event is emitted (i.e., when one presses an arrow key) the MARS server generates two input messages, one to the opponent and other to the same device. The event `FIRE` from any device is mapped to the event `SHOT` in both devices (lines 16–17 and 21) but here we use a filter function (line 24–30). Such function sets the first argument of the event `SHOT` to **`true`**, if the source and target devices are the same, or **`false`** otherwise.

```
1  local player1 = nil
2  local player2 = nil
3
4  MARS.onConnect = function (p)
5    if player1 == nil then
6      player1 = p;
7    else
8      player2 = p;
9
10     map (player1, "MY_POS", player2, "OPPONENT_POS")
11     map (player2, "MY_POS", player1, "OPPONENT_POS")
12
13     map (player1, "UPDATE_POS", player2, "OPPONENT_MOVE")
14     map (player2, "UPDATE_POS", player1, "OPPONENT_MOVE")
15
16     map (player1, "FIRE", player2, "SHOT", tansfFunc)
17     map (player2, "FIRE", player1, "SHOT", tansfFunc)
18   end
19
20   map (p, "UPDATE_POS", p, "MY_MOVE")
21   map (p, "FIRE", p, "SHOT", transfFunc)
22 end
23
24 function tansfFunc (from, to, args)
```

```
25   if from == to then
26     return true, {true, args[1], args[2]}
27   else
28     return true, {false, args[1], args[2]}
29   end
30 end
```

Listing 6.10: Mapping script of the shooting game.

## 6.4 Use Case 3: Video Wall

Bob wants to present in a multi-monitor setup the new video advertisement he has been working on in the last months. Because this video is too large and Bob has only regular computers without much computational power, he sets up the environment in a way that each computer is connected to a single monitor and it processes only part of the video. Bob wants that during the presentation all monitors work as if they were connected to a single computer, showing the video in-sync. Additionally, Bob also has a controller that he can use to seek the video whenever one asks to (also adapted from the Timing Object W3C draft spec [96]).

Our implementation of this use case has two Céu programs, one that runs on devices connected to a monitor, and other that runs on a controller device. The former (Listing 6.11) only presents the video on the monitor, and the latter (Listing 6.12) reacts to inputs events and controls the video wall.

```
1  input (none) START;
2  input (int)  SEEK;
3
4  var double crop_x;
5  var double crop_y;
6  var double crop_w;
7  var double crop_h;
8
9  (crop_x, crop_y, crop_w, crop_h) = <get values from command line>;
10
11 await START;
12
13 var&? Scene scene = spawn Scene(<...>);
14 watching scene
15   loop do
16     var&? Play video = spawn Play (<...>);
17     call Player_Set_Double (&video, "crop-left",   crop_x);
18     call Player_Set_Double (&video, "crop-top",    crop_y);
19     call Player_Set_Double (&video, "crop-right",  crop_w);
20     call Player_Set_Double (&video, "crop-bottom", crop_h);
21
22     par/or do
23       await video;
24     with
25       var int position;
26       every (position) in SEEK do
27         call Player_Seek (&video, position);
```

```
28        end
29      end
30    end
31  end
```

Listing 6.11: CÉU source code of program that presents part of the video in a given monitor.

```
1  output (int) COMMAND;
2
3  var&? Scene scene = spawn Scene(<...>);
4  watching scene
5    var char key;
6    every key in CM_SCENE_KEY do
7      if key == "space" then
8        emit COMMAND (0);
9      else/if is_a_number (key) then
10       emit COMMAND (to_number(key));
11     end
12   end
13 end
```

Listing 6.12: CÉU source code of the video wall controller.

The program that presents the video (Listing 6.11) has two input events: `START` and `SEEK` (lines 1–2). This code assumes that the region of the video the program should present is passed as command line arguments, whose values are stored in the variables starting with the prefix `crop_` (lines 4–9). The **`await`** in line 11 makes the program to halt until it receives the event `START`. From this point, it creates a `Scene` and a cropped video following the values passed as argument (lines 13–20). At this point, the **`par/or`** composition creates two trails (lines 22–29). The first only waits the video to end (line 23). The second reacts to occurrences of the event `SEEK`, seeking the video to the proper position. The surrounding **`loop`** statement (lines 15–30) makes the program to start the video again whenever it ends, presenting the advertisement in a loop.

The controller program (Listing 6.12) is simple and straightforward, having only one output event (`COMMAND`). It just reacts to key inputs (lines 6–11) and emits that event passing the value 0 if the key pressed is "space", or it passes an integer value if the key pressed is a number.

Listing 6.13 presents the interface table of this application. It has two interfaces: `VIDEO` and `CONTROLLER`. The first has two input events, `START` and `SEEK`. The second one output event, `COMMAND`.

```
1  interfaces {
2    VIDEO = {
3      inputs = {
4        START = {}, SEEK = {"int"}
5      }
6    },
7    CONTROLLER = {
8      outputs = {
9        COMMAND = {"int"}
```

```
10      }
11    }
12  }
```

Listing 6.13: Interfaces table of the video wall application..

And the mapping script is depicted in Listing 6.14. It stores the video instances in the table `videos`, and the controller in the variable `controller`. This script implements the `onConnect` callback, mapping the event `COMMAND` from the controller to the events `START` and `SEEK` of the videos. The filter functions define when these mappings should be triggered: if the argument of the event `COMMAND` is 0, then the event `START` is triggered and the `SEEK` is not, otherwise the event `SEEK` is triggered and the `START` is not.

```
1  local controller = nil
2  local videos = {}
3
4  MARS.onConnect = function (p)
5    local interfaces = p:getInterfaces ()
6
7    for i,_ in pairs (interfaces) do
8      if i == "CONTROLLER" then
9        controller = p
10       for _, v in pairs (videos) do
11         map (controller, "COMMAND", v, "START", transfStart)
12         map (controller, "COMMAND", v, "SEEK", transfSeek)
13       end
14     elseif i == "VIDEO" then
15       table.insert (videos, p)
16       if controller ~= nil then
17         map (controller, "COMMAND", p, "START", transfStart)
18         map (controller, "COMMAND", p, "SEEK", transfSeek)
19       end
20     end
21   end
22 end
23
24 function transfStart (from, to, args)
25   return args[1] == 0, nil
26 end
27
28 function transfSeek (from, to, args)
29   return not args[1] == 0, {args [1]}
30 end
```

Listing 6.14: Interfaces table of the video wall application..

## 6.5 Discussion

Mars programming model separates the concerns when developing distributed interactive applications in two phases: the development of the application logic (Céu source code) and the definition of how programs communicate (mapping script). Note that these phases can be carried out in

any order (or even in parallel) or by different programmers. In fact, the same CÉU application may be used with different mapping scripts due to the loose coupling among these two codes.

Besides this programming model, we have explored the MARS consistency guarantees for implementing these use cases. For instance, consider the first application (Social Viewing and Media Control). If the "togetherness" is enabled, whenever Alice or Bob pauses the video, both presentations will pause at exactly same frame. This may be useful in the scenario considered, in which they are in separate environments and may pause the video to discuss a specific scene.

Still in the first use case, the lack of perfect playout synchronization usually is not a problem for social viewing applications. Geerts et al. [98] report in a study that users communicating using voice while watching the same content together start to notice synchronization problems only above 2 seconds delays. When considering users using chat applications, the difference is noticeable above 4 seconds delays.

The second use case (Online Education) also explores MARS consistency property. Even when multiple students request at the same time the control over the video, there is no divergence regarding to whom Alice has granted the control. Regarding the synchronization of playouts, the same reasoning of the first use case can be applied in this scenario. That is, because students and the teacher are geographically apart and communicate using online tools, some differences in the playback is acceptable and does not compromise the overall experience.

In [94], Mauve has discussed some issues in distributed virtual environments induced by the absence of consistency guarantees. One of the examples he cites in that work is the *dead man that shoots*, which is illustrated by an action game whose state of the avatars are different in each device due to network delays and jitter. This situation may lead to a scenario that an avatar, which is dead to a given player, shoots and kills another avatar.

The implementation of the third use case in MARS is an alternative for this type of game that does not have this problem. Even if both avatars shoot each other nearly at the same time, all players will receive event `SHOT` in the same order and will therefore reach the same final state. This property will hold even if we change our implementation to a multiplayer game having more than two players. Again, the MARS consistency guarantees is vital for assuring that no device stays in a state that is different from others.

However, in this application we have a responsiveness penalty due to the communication protocol implemented in MARS. When a player presses one of

the arrow keys or space, the corresponding action is not immediately reflected on the game. Instead, the Mars runtime sends an output message to the server, that computes whether this message should trigger an input event in other devices, to then send the appropriate input messages. Furthermore, the synchronization of input events discussed in Section 5.5 introduces an additional delay for generating the events that update the game. Thus, even though Mars guarantees the consistency for this application, its communication protocol may hinder the "playability" of the game.

Now let's discuss the last application. Mars guarantees that all monitors start and seek at the same time, but it does not provide any support for playout synchronization. Thus, because devices run at their own pace, it is likely that the video wall does not show the video in-sync. An alternative to bypass this problem is for the controller to generate an event for each frame. That is, the program continuously waits for a given event to then change the frame being presented. Because of the Mars input events synchronization property, all devices would change frames in-sync.

There are some problems with this alternative approach. For this discussion, consider a video with frame rate of 30 fps (*frames per second*). To exhibit each frame (i.e., at each 33ms) the controller sends a message to the server, that then sends a message to each device, leading to a frame changing. However, because the server adds an offset to each input event to compensate for network delays, the 30fps frame rate would not be possible to be maintained.

There is a conceptual problem with this approach. According to the zero-delay synchronous hypothesis that Céu relies on, reactions are conceptually instantaneous. In practice, the synchronous hypothesis holds if reactions execute faster than the rate of incoming events [17]. Typically, a program takes some milliseconds to compute the reaction to an event, which is the same order of magnitude of incoming events if we generate an event for each frame. That is, this scenario violates the synchronous hypothesis, leading the program to continuously accumulate delays between occurrences and reactions to events [17]. This rationale explains why Mars is not designed for massive distributed applications in which the message exchanging rate is high.

In sum, the development of these use cases indicates that our approach can be used for programming real-world distributed applications. From the point of view of programming distributed interactive multimedia applications, the programming model and guarantees of Mars are handy for implementing use cases in which consistency is required. To the best of our knowledge, there is no proposal in this field that provides this consistency guarantee without requiring programmers to explicitly implement it. However, our approach is

not a silver bullet for programming all interactive multi-device applications, as it fails to support the execution of use cases that require fine-grained synchronization or have a high rate of interactions.

# 7
# Related Work

In this work we have explored the use of the synchronous hypothesis for supporting the development and execution of multi-device interactive multimedia applications and we have proposed a programming model for developing these applications. In this chapter we revisit some related works and compare them with our proposal.

Most of the recent work within the multimedia community approaching multi-device applications focuses on the infrastructure for supporting their execution with certain guarantees, disregarding the programming support at language level. Distributed multimedia synchronization is one of the most tackled problems in this domain in the last decade. Some of the approaches for this problem have proposed extensions to transport protocols [99, 100, 101, 102], network-level techniques [103, 104, 105, 106], audio fingerprinting-based synchronization [107, 108, 109] or adaptation of media playout [110, 111, 112]. Even though some of these works represent an advance in the state of the art of distributed multimedia synchronization, none of them is concerned with proposing programming abstractions for supporting the development of distributed applications. That is, their main targets are system developers rather than application developers. Besides, most of these techniques do not consider user interactivity.

Our work in multi-device applications does not approach the distributed multimedia synchronization problem. Our main focus is on supporting programmers in developing these applications considering interactivity. In fact, Mars consistency approach was designed exactly to solve problems that arise from the interaction of users with applications. However, our work could be enhanced by implementing some of those proposals for distributed synchronization, which is a point that we left for future work as discussed in next the chapter.

Recent advances in the digital TV industry has leveraged research in the multi-device applications landscape. Three of the most prominent digital TV systems in the world are ATSC (North America) [113], DVB-T (European-based) [114] and ISDB-Tb (Japan-based, but adopted mostly in Latin America) [13] and they all support multi-device interactive multimedia

applications (often called companion or second screen applications).

The ATSC system specification defines a communication protocol between a primary device (usually TVs or set-top boxes) and secondary devices (e.g, laptops, tablets and smartphones). This protocol supports the following main features: automatically launch an application in secondary devices from a primary device, application to application communication, and service discovery. The DVB system also supports these features, but it goes a step further and defines a protocol for clock synchronization between the primary and secondary devices for applications that require fine-grained synchronization.

Both ATSC and DVB do not restrict the language in which companion applications should be programmed, but rather they define a set of protocols that compliant (web-based or native) applications should adhere to. The main advantage of this design is that programmers are not tied to a single technology for developing multi-device applications. However, this characteristic also prevents the system to ensure further guarantees, such as determinism. Even though it is possible to develop deterministic applications to run on ATSC and DVB systems (e.g., programmatically enforcing the determinism or using a deterministic language) one cannot assume that all companion applications have this property. Likewise, the specifications of both systems do not impose any consistency guarantees, which should be implemented at application-level when needed. Furthermore, developers should either implement the communication protocol or use a compliant communication library so applications can properly exchange messages.

The ISDB-Tb system adopts NCL and Lua as languages for application development. Unlike ATSC and DVB, the ISDB-Tb defines a communication API (instead of a protocol) for both languages. From the programming perspective, it means that developers can use this API for communicating with other applications. Internally, the system is responsible for implementing this API and the devices' communication layer. Thus, even though developers still use communication primitives in their codes, they need not worry about implementing lower-level communication protocols. Regarding determinism, as discussed throughout this thesis the NCL language has some non-deterministic constructs, therefore we cannot say that the ISDB-Tb system can guarantee this property. Regarding consistency, similar to ATSC and DVB, the system specification does not impose it, making it an implementation-dependent feature.

In our approach, we can ensure not only the ordering of messages (sequential consistency) but also that each application always behaves deterministically because we rely on a deterministic language. Additionally, we have

designed MARS in a way that the use of explicit communication primitives in source codes is unnecessary—in fact, it is discouraged because the runtime may not be able to guarantee the total ordering in this case.

In the web panorama, there have been some recent efforts promoting distributed (in this context, called *cross-device*) applications. Cross-device are web-based applications whose user interfaces (UIs) are designed to be rendered on multiple devices, in which each device acts either as a mirror or renders part of a larger interface. Most of the research in this area focuses on defining languages [115, 116, 117], frameworks [118, 119, 120, 121], and development tools [122, 123, 124, 125] for supporting cross-device applications.

We can say that cross-device applications fall into Levin's complementary category [18] (applications that complement one another creating an experience as a connected group), therefore consistency is a concern. Devices running part of a distributed UI share data whose state must be synchronized. In literature, there are centralized [118, 120, 126] and decentralized [127, 128, 129] approaches for handling consistency in cross-device applications.

Despite sharing some similarities, there are some differences between the type of applications we target in this work and cross-device web-based UIs. We approach distributed interactive multimedia applications, in which timing is a crucial aspect. On the other hand, even though the number of video- and audio-based applications/services has dramatically increased on the web in the last two decades (e.g., Spotify, YouTube, Google Play Music, Amazon Music, Dailymotion, etc.) web pages, in general, are still atemporal. Thus, the consistency techniques implemented by works targeting cross-device applications are mainly concerned in guaranteeing data consistency disregarding timing constraints. As pointed by Mauve et al. [95], consistency in systems that target continuous objects is not just about defining a global order of events, but also about guaranteeing that each operation is executed at the correct point in time. In our work we have considered this issue when designing the MARS consistency algorithm.

Although not common, there are some timing sensitive use cases on the web (e.g., timing sensitive Twitter widget that can replay timestamped tweets). Arntzen et al. [96] have proposed the web timing object JavaScript API for supporting those use cases. The main goal of that work is to encapsulate the complexity of clock synchronization across devices into an object that implements a synchronized and shared timeline (aka timing object).

The timing object work is not directly related to ours, but we have cited it here for completeness. This JavaScript API was designed specifically for implementing a synchronized timeline, that is, it *per se* does not provide any

consistency guarantee. However, combining the timing object with frameworks for cross-device applications can enable the development of consistency approaches that consider timing constraints, similarly to our work.

Collaborative Virtual Environments (CVEs) have some characteristics similar to applications we are targeting. A CVE is a system in which multiple users are immersed in a virtual world visualizing and interacting with shared objects [130]. Massively multi-player or serious games, large-scale virtual cities, and open space military training are examples of CVEs [131]. In a CVE application, users' updates must be propagated to others with low-latency maintaining the system responsive, otherwise participants may become frustrated [132] and even lose interest in the application [133]. Additionally, the system should guarantee that those updates are applied in each device consistently because users may be interacting with the same shared object concurrently. And a third requirement of these applications is scalability, as CVEs are generally designed to scale for hundreds or thousands of users.

Dead-reckoning [134] is a classic technique often used in CVEs for tackling the consistency problem. It is based on a combination of state prediction and state transmission. As each device "knows" how shared objects behave over time without users' interaction, they can predict locally the position of those objects as the time passes. When the state of an object changes in a device in a way that prediction of positions of that object in other peers become inconsistent, an update message is issued to all devices. Upon receiving this message, devices update the state of that object and use this new state in future predictions. Mauve et al. [95] have demonstrated that this technique cannot prevent continuous applications to reach consistent, but incorrect states, such as the distributed driver instructor application described in Chapter 5 (page 80).

In that same work [95], the authors have proposed an approach that provides consistency for continuous applications based on two complementary techniques: local-lag and timewarp. Local-lag consists in decreasing the system responsiveness by delaying the execution of actions for a certain amount of time. Consider a distributed setting with 3 devices. If device $D_1$ generates the operation $O_1$ at timestamp $t_1$, then the system adds the timing offset $t^*$ to $O_1$ (i.e., the timestamp for that operation will be $t_1 + t^*$) and sends this operation to $D_2$ and $D_3$. All three devices should execute $O_1$ when their local clock reaches the time $t_1 + t^*$. The authors also propose a method based on the maximum average of network delays for determining a minimal value to $t^*$ so the probability of all devices receive the message before the time $t_1 + t^*$ is high, without compromising the system responsiveness.

Because it is likely that eventually messages arrive at devices after the operation timestamp has passed, the authors complement the local-lag technique with timewarp. Timewarp is a mechanism for recovering consistency when network delay and/or jitter are greater than the initial estimate. It consists in storing in a list, ordered by timestamp, the last N operations received. When a new operation arrives, the system adds it to the list and computes whether its current state is consistent or not: the state is consistent if that insertion occurs in the last position (or in the first, depending on the sorting order) of the list. If an inconsistency is detected, the system rolls back its state until the last known consistent state and reapplies all the operations from that point and on. The authors prove that the timewarp algorithm has the complexity $O(n^2)$, where $n$ is the number of participants in a session.

Our consistency approach has some similarities with the local-lag technique, as both add a timing offset (based on network delay estimates) to operations for executing them synchronously some time later in all devices[1]. However, there are some fundamental differences between our work and that one. First, Mauve's approach is decentralized, while Mars's is centralized, which means that the former is more robust to failures. Second, their approach assume a synchronized clock between devices, otherwise the maximum offset between any two clocks should be added to the $t^*$ value in the local-lag algorithm. Mars adopts the GALS style in which there is no assumption regarding clock synchronization. Third, their work admits some short-term inconsistencies (which are repaired by the timewarp algorithm) so the responsiveness of systems is not compromised. Mars favors consistency rather than responsiveness, halting applications when messages experiment unusual network delays. In sum, the Mauve's approach is more suitable for applications in which peers' clocks are synchronized (or the maximum clock drift between devices are known) and that can tolerate some short-term inconsistencies in favor of responsiveness. In contrast, our approach is more applicable when there is no assumption regarding clocks synchronization and the system's consistency should be maintained even if it means decreasing its responsiveness.

The approach of halting the execution of an application to prevent inconsistencies is also used in the bucket synchronization algorithm implemented in the game Age of Empires [136]. This algorithm is based on executing actions in a lock-step way on all clients. The communication timeline is divided into frames (or buckets) of fixed length (set to 200ms in the authors' work). Inputs gathered in a given frame are applied two frames after that. If a given device

[1] It worth mentioning that delaying the execution of an operation is not a novelty from neither of these works (see [135]).

experiences the end of a frame before it has received the next one, the game pauses until the expected frame arrives. At this point, a routine to adjust the frame length starts to execute.

That work is an evidence that both, delaying the execution of actions in response to user interactions and halting applications to avoid inconsistencies, are feasible approaches in real-world applications. The responsiveness requirements in multiplayer games such as Age of Empires are more strict than the ones of applications we target in our work. Despite that, there is no indication that the bucket synchronization approach compromise the user experience of that game.

Among other consistency techniques, the remote lag [137] and local perception filters [138, 139] are two of the most commonly used in real-world games. These algorithms are mainly concerned in increasing the game's responsiveness while admitting some degree of inconsistencies, unlike the MARS's consistency approach that does not accept them.

To the best of our knowledge, there is no work in literature that targets distributed interactive multimedia applications guaranteeing deterministic behaviors for local applications and consistency, considering timing constraints of multimedia (continuous) applications, for the whole system. Furthermore, few works in literature approach the problem of supporting the development of these applications at language-level. And the works that address such a problem are not concerned in promoting a programming model in which programmers do not have to explicit use communication primitives.

# 8
# Conclusion

In this thesis we have approached the problem of supporting the development and execution of distributed interactive multimedia applications using Céu. Our work can be divided into two complementary proposals: first, the use of a synchronous language for approaching local applications, and second the instantiation of a GALS architecture for distributed applications. Therefore, in essence our work is about investigating whether the synchronous hypothesis and the GALS design are suitable in this domain.

Regarding local applications, we already had evidences that synchronous languages could be used for low-level multimedia processing, without considering any user interactivity, especially due to the existence of languages such as ChucK and Pure Data. More recently, the development of Smix has pointed out the feasibility of designing a synchronous domain specific language for high-level multimedia programming that supports interactive applications. Our work is partially inspired by those and it studies how well the general-purpose synchronous language Céu can be used in this domain.

One of the main results of this thesis is the indication that Céu is suitable for multimedia. Our study has covered both aspects, viz. syntactic and semantic. Syntactically, we have managed to directly implement the causal operators of the Interval Expression model using Céu constructs, that is, Céu can express the most common causal relationships between media objects. Semantically, we have concluded that the synchronous execution model and the semantics of the language avoid non-determinism, even in interactive multimedia applications, and can enforce intermedia synchronization. We believe that language creators can benefit from these results and consider the use of the synchronous hypothesis when designing or evolving existing multimedia languages.

Céu-Media, one of the practical results of this thesis, reifies the development of deterministic multimedia applications in Céu. By ruling the presentation clock to the program's logical time, it provides frame-level synchronization accuracy enforced by the language's execution model. That is, the synchronization of media objects in a Céu-Media application is enforced at language-level.

The main drawback of this approach focused on correctness is it possible impact on user experience. When the drift between the real-world and the logical time increases, the general presentation frame rate drops, which slows down the video or leads to audio glitches. However, from the synchronization perspective, all objects are presented at the correct (logical) time.

The limitations of the synchronous model became more evident when we moved from local to distributed applications. We wanted to keep taking advantage of Céu and Céu-Media, but using a global synchronous clock in a network without timing guarantees is impracticable. The use of GALS in this context was our approach to explore the synchronous execution model locally, while asynchronously exchanging messages. In fact, the assumptions we have made regarding the underyling network infrastructure and the use of a synchronous language necessarily led us to the GALS architectural style.

However, GALS *per se* does not guarantee consistency. Thus, we presented a consistency algorithm for Mars that guarantees that all intended devices react at the same logical time to input events sent by the server. As discussed in Chapter 5, consistency in distributed multimedia applications is not only about messages ordering, but also about executing actions at the correct time. We could not find any work in the multimedia literature that provides consistency with timing guarantees in a network with no bounded delay or without relying on clock synchronization.

Mars consistency model focuses on correctness and has as drawback its possible impact on user experience, similarly to Céu-Media. When we admit that applications may halt due to late messages, users may be affect by this design. Defining a good RTT estimate helps to avoid recurrent application haltings, but this is out of the scope of this thesis.

Another result of our work is the programming model that separates concerns during the development of distributed applications. Conceptually, there are two actors involved when programming a Mars application: one that programs the application logic and other that defines the inter-application communication bindings. This approach allows existing regular Céu codes to be compiled *as is* using the Mars compilation process, generating programs ready for joining a distributed session. The same program may also be executed in Mars sessions using different mapping scripts, indicating the flexibility of this programming model.

Our centralized architecture has some drawbacks. First, there is the single point of failure problem, that is, if the server fails the whole session ends. It also limits the scalability of Mars, which is not able to satisfactorily handle dozens of simultaneous clients. Because the server must be initiated with a

mapping script, it should be restarted each time one wants to run a different application in the same network, which means that each application requires a dedicated server to coordinate the session.

We point as the main weakness of our approach not meeting all requirements of use cases in Chapter 6. The communication delay breaks the synchronous hypothesis in the distributed scenario, leading to an offset between the generation of events and their actual processing on devices. This hinders achieving distributed synchronization as demanded in some use cases. And increasing message exchanging rate to create several synchronization points does not work well because this also violates the synchronous hypothesis. That is, this problem is intrinsic to GALS systems.

In sum, we highlight the following points as main contributions of this thesis:

- A study about the suitability of Céu for programming multimedia applications covering syntactic and semantic aspects;
- An approach based on Céu for guaranteeing deterministic executions and frame-level synchronization accuracy enforced at language-level;
- The implementation of a consistency model for distributed applications that guarantees that all devices process messages in the same order and at the same time.

## 8.1 Future Works

There are several future works that can extend this thesis. Some of them are:

- To extend Céu-Media programming model for compositionality. A Céu-Media `Scene` does not provide a full-fledged composition feature, as it cannot be added recursively to other `Scene`s. There are some different design choices for this feature. For instance, a `Scene` could be dynamically moved to another, which would result in all objects switching windows while maintaining their states. In this case, how to proper synchronize these objects in the current Céu-Media execution model is a question that is not trivially answered.
- To extend Céu-Media for handling novel media modalities. Recently, part of the multimedia research community is investigating the integration of novel media modalities, such as olfactory, haptic, and thermoceptic with traditional audiovisual content. The synchronization require-

ments of these modalities are still under investigation. Extending Céu-Media to integrate these modalities with the ones already supported can help to better understand these requirements.

– To investigate non-GALS architectures for multimedia. If one uses a network with timing guarantees, it is possible to use the synchronous execution model with different distribution approaches. For instance, if the maximum delay is negligible, one can design a distributed synchronous system having a global common clock, which would facilitate the development of distributed synchronization. However, one has to deal with other problems such as distributed consensus.

– To decentralize Mars. There are some practical implications if one decentralizes Mars and tries to provide the same guarantees described in this work. The first is to design a proper consistency algorithm, which could be based on Lamport's logical clock. Another challenge is to compute the correct timing offset of messages to guarantee that devices react synchronously to input events.

– To improve the robustness of the system. As Mars has a single point of failure, the system depends on the server to work. One could implement replication approaches for replacing the server if it fails or even select one of the clients to be the new server.

– To experiment weaker consistency models. Our study has indicated that the sequential consistency model ensures the guarantees we were interested in providing to applications. One could investigate whether weaker consistency models, such as causal or eventual consistency, can provide similar guarantees without disregarding the timing aspect.

# Bibliography


[1] P. Teehan, M. Greenstreet, and G. Lemieux, "A Survey and Taxonomy of GALS Design Styles," *IEEE Design & Test of Computers*, vol. 24, no. 5, pp. 418–428, sep 2007.

[2] A. Duda, C. Erif Keramane, and C. Keramane, "Structured Temporal Composition of Multimedia Data," in *Proceedings of the International Workshop on Multi-Media Database Management Systems*. IEEE Computer Society, aug 1995, p. 136.

[3] G. F. Lima, R. C. M. Santos, and R. G. de Albuquerque Azevedo, "Programando aplicações multimídia no gstreamer." Teresina, Brasil: SBC, 11 2016.

[4] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sept 1979.

[5] M. Louvel, P. Bonhomme, J. P. Babau, and A. Plantec, "A network resource management framework for multimedia applications distributed in heterogeneous home networks," in *2011 IEEE International Conference on Advanced Information Networking and Applications*, March 2011, pp. 724–731.

[6] J. Jin and K. Nahrstedt, "Qos specification languages for distributed multimedia applications: a survey and taxonomy," *IEEE MultiMedia*, vol. 11, no. 3, pp. 74–87, July 2004.

[7] S. Laplace, M. Dalmau, and P. Roose, "Kalinahia: Considering quality of service to design and execute distributed multimedia applications," in *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, April 2008, pp. 951–954.

[8] M. Sarkis, C. Concolato, and J.-C. Dufourd, "The virtual splitter: Refactoring web applications for themultiscreen environment," in *Proceedings of the 2014 ACM Symposium on Document Engineering*, ser. DocEng '14. New York, NY, USA: ACM, 2014, pp. 139–142.

[9] H. V. Hansen, F. Velázquez-García, V. Goebel, and T. Plagemann, "Efficient data sharing for multi-device multimedia applications," in *Proceedings of the Workshop on Multi-device App Middleware*, ser. Multi-Device '12. New York, NY, USA: ACM, 2012, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2405172.2405174

[10] M. Levin, *Designing Multi-Device Experiences: an ecosystem approach to user experiences across devices*, first edit ed., M. Treseler, Ed. O'Reilly Media, 2014.

[11] J. Preißinger and T. Landes, "Realizing consistent event ordering in distributed shared memory systems," 6 2006, tobias Landes and Jörg Preißinger. Realizing Consistent Event Ordering in Distributed Shared Memory Systems. In Hamid R. Arabnia, editor, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'2006, pages 10-16, Las Vegas, NV, June 2006.

[12] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[13] ABNT, "NBR 15606-2:2011 "Digital Terrestrial Television - Data Coding and Transmission Specification for Digital Broadcasting - Part 2: Ginga-NCL for Fixed and Mobile Receivers - XML Application Language for Application Coding," São Paulo, Brazil, Tech. Rep., 2011.

[14] W3C, "Synchronized Multimedia Integration Language (SMIL 3.0)," 2008, W3C Recommendation https://www.w3.org/TR/smil/.

[15] ——, "HTML5 - A vocabulary and associated APIs for HTML and XHTML," 2014, W3C Recommendation. http://www.w3.org/TR/html5/.

[16] E. Lee, "The Problem with Threads," *Computer*, vol. 39, no. 5, pp. 33–42, may 2006.

[17] R. C. M. Santos, G. F. Lima, F. Sant'Anna, R. Ierusalimschy, and E. H. Haeusler, "A memory-bounded, deterministic and terminating semantics for the synchronous programming language céu," in *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2018. New York, NY, USA: ACM, 2018, pp. 1–18. [Online]. Available: http://doi.acm.org/10.1145/3211332.3211334

[18] Real Racing 2 HD Update Enables Wireless Splitscreen Multiplayer On iPhone 4S And iPad 2 (video). Accessed: 2018-03-05. [Online]. Available: https://www.geeky-gadgets.com/real-racing-2-hd-update-enables-wireless-splitscreen-multiplayer-on-iphone-4s-and-ipad-2-video-06-10-2011/

[19] Use Your Smartphone as a Remote Control For Your TV. Accessed: 2018-03-05. [Online]. Available: https://techpiration.com/use-your-smartphone-as-a-remote-control-for-your-tv/

[20] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[21] J.-p. T. Dumitru Potop-butucaru, Robert De Simone, "The synchronous hypothesis and synchronous languages," in *Embedded Systems Handbook*. CRC Press, 2005.

[22] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.

[24] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1998.

[25] F. Sant'anna, R. Ierusalimschy, N. Rodriguez, S. Rossetto, and A. Branco, "The Design and Implementation of the Synchronous Language CÉU," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, pp. 98:1–98:26, Jul. 2017.

[26] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau, "Assessing task migration impact on embedded soft real-time streaming multimedia applications," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, p. 518904, Nov 2007. [Online]. Available: https://doi.org/10.1155/2008/518904

[27] C. Fan, "Realizing a soft real-time framework for supporting distributed multimedia applications," in *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Aug 1995, pp. 128–134.

[28] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, Dec 1998, pp. 4–13.

[29] R. Steinmetz, "Analyzing the multimedia operating system," *IEEE MultiMedia*, vol. 2, no. 1, pp. 68–84, Spring 1995.

[30] G. F. Lima, "A synchronous virtual machine for multimedia presentations," Ph.D. dissertation, Department of Informatics, PUC-Rio, Rio de Janeiro, RJ, Brazil, 2015.

[31] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter, "A Survey on Reactive Programming," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, pp. 1–34, 2013.

[32] M. Papathomas, G. S. Blair, G. Coulson, P. Robin, and D. Multimedia, "Addressing the real-time synchronization requirements of multimedia in an object-oriented framework," in *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 2417, 1995, pp. 1–12.

[33] G. S. Blair, M. Papathomas, G. Coulson, P. Robin, J. B. Stefani, F. Horn, and L. Hazard, "Supporting real-time multimedia behaviour in open distributed systems: an approach based on synchronous languages," *ACM Multimedia'94*, pp. 299–306, 1994.

[34] J. M. Eyzell and J. Farines, "Using ESTEREL for building synchronization mechanisms in multimedia systems," *IEEE Conference on Protocols for Multimedia Systems - Multimedia Networking, 1997*, pp. 269–272, 1997.

[35] G. S. Blair, G. Coulson, M. Papathomas, P. Robin, J. B. Stefani, F. Horn, and L. Hazard, "A programming model and system infrastructure for real-time synchronization in distributed multimedia systems," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 1, pp. 249–263, 1996.

[36] P. Hoepner, "Presentation scheduling of multimedia objects and its impact on network and operating system support," *Network and Operating System Support for Digital Audio and Video*, pp. 132–143, 1991.

[37] G. Wang and P. R. Cook, "On-the-fly programming: Using code as an expressive musical instrument," in *Proceedings of the 2004 Conference on New Interfaces for Musical Expression*, ser. NIME '04. Singapore, Singapore: National University of Singapore, 2004, pp. 138–143. [Online]. Available: http://dl.acm.org/citation.cfm?id=1085884.1085915

[38] M. Puckette, "Pure data: another integrated computer music environment," in *in Proceedings, International Computer Music Conference*, 1996, pp. 37–41.

[39] B. Vercoe, *Csound: A Manual for the Audio Processing System and Supporting Programs with Tutorials*. Massachusetts Institute of Technology, 1992.

[40] Y. Orlarey, D. Fober, and S. Letz, "FAUST: An efficient functional approach to DSP programming," in *New Computational Paradigms for Computer Music*, 2009.

[41] McCartney, James, "SuperCollider: a new real time synthesis language," in *Proceedings of the 1996 International Computer Music Conference*, 1996.

[42] M. D. Natale and A. L. Sangiovanni-Vincentelli, "Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, April 2010.

[43] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, and D. D. Cofer, "A Formal Architecture Pattern for Real-Time Distributed Systems," in *2009 30th IEEE Real-Time Systems Symposium*, Dec 2009, pp. 161–170.

[44] D. Potop-Butucaru, A. Azim, and S. Fischmeister, "Semantics-preserving implementation of synchronous specifications over dynamic TDMA distributed architectures," in *Proceedings of the tenth ACM international conference on Embedded software - EMSOFT'10*. ACM Press, 2010. [Online]. Available: https://doi.org/10.1145/1879021.1879048

[45] M. D. Natale, Q. Zhu, A. Sangiovanni-Vincentelli, and S. Tripakis, "Optimized implementation of synchronous models on industrial LTTA systems," *Journal of Systems Architecture*, vol. 60, no. 4, pp. 315–328, apr 2014. [Online]. Available: https://doi.org/10.1016/j.sysarc.2014.01.003

[46] A. Girault and C. Ménier, "Automatic production of globally asynchronous locally synchronous systems," in *Embedded Software*. Springer Berlin Heidelberg, 2002, pp. 266–281. [Online]. Available: https://doi.org/10.1007/3-540-45828-x_20

[47] K. Sun, L. Besnard, and T. Gautier, "Optimized distribution of synchronous programs via a polychronous model," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, oct 2014. [Online]. Available: https://doi.org/10.1109/memcod.2014.6961842

[48] G. Delaval, A. Girault, and M. Pouzet, "A type system for the automatic distribution of higher-order synchronous dataflow programs," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems - LCTES'08*. ACM Press, 2008. [Online]. Available: https://doi.org/10.1145/1375657.1375672

[49] B. Aminof, S. Rubin, I. Stoilkovska, J. Widder, and F. Zuleger, "Parameterized model checking of synchronous distributed algorithms by abstraction," in *Lecture Notes in Computer Science*. Springer International Publishing, dec 2017, pp. 1–24. [Online]. Available: https://doi.org/10.1007/978-3-319-73721-8_1

[50] S. Chaki and J. Edmondson, "Toward parameterized verification of synchronous distributed applications," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, ser. SPIN 2014. New York, NY, USA: ACM, 2014, pp. 109–112. [Online]. Available: http://doi.acm.org/10.1145/2632362.2632368

[51] W. Steiner and J. Rushby, "TTA and PALS: Formally verified design patterns for distributed cyber-physical systems," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 1–15, 2011.

[52] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[53] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and PC, "PALS: Physically asynchronous logically synchronous systems," University of Illinois at Urbana Champaign, Tech. Rep., 2009. [Online]. Available: http://hdl.handle.net/2142/11897{%}0A

[54] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to scade/lustre to tta: A layered approach for distributed embedded applications," in *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, ser. LCTES '03. New York, NY, USA: ACM, 2003, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/780732.780754

[55] M. Gunzert, "Building safety-critical real-time systems with synchronous software components," *IFAC Proceedings Volumes*, vol. 32, no. 1, pp. 63 – 68, 1999, 24th IFAC/IFIP Workshop on Real Time Programming WRTP 99, Schloss Dagstuhl, Germany, 30 May - 3 June. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667017399664

[56] A. Al-Nayeem, C. Kim, W. Kang, P. L. Wu, and L. Sha, "Middleware design for physically-asynchronous logically-synchronous (pals) systems," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, Sept 2013, pp. 1–10.

[57] J. Park and T. Kim, "A method of logically time synchronization for safety-critical distributed system," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, Jan 2016, pp. 356–359.

[58] B. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture: an overview," Internet Requests for Comments, RFC Editor, RFC 1633, June 1994, http://www.rfc-editor.org/rfc/rfc1633.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1633.txt

[59] R. J. Vetter, "Atm concepts, architectures, and protocols," *Commun. ACM*, vol. 38, no. 2, pp. 30–ff., Feb. 1995. [Online]. Available: http://doi.acm.org/10.1145/204826.204831

[60] A. Benveniste, A. Bouillard, and P. Caspi, "A unifying view of Loosely Time-Triggered Architectures *," in *International Conference on Embedded Software International Conference on Embedded Software*, 2010, pp. 189–198.

[61] B. Rajan and R. K. Shyamasundar, "Multiclock Esterel: a reactive framework for asynchronous design," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, ser. IPDPS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 201–209.

[62] G. Berry, S. Ramesh, and R. K. Shyamasundar, "Communicating Reactive Processes," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93. New York, NY, USA: ACM, 1993, pp. 85–98.

[63] S. Ramesh, "Communicating Reactive State Machines: Design, Model and Implementation," *IFAC Proceedings Volumes*, vol. 31, no. 32, pp. 105–110, sep 1998.

[64] F. Jebali, F. Lang, and R. Mateescu, "Formal modelling and verification of GALS systems using GRL and CADP," *Formal Aspects of Computing*, vol. 28, no. 5, pp. 767–804, sep 2016.

[65] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Computer Languages, Systems and Structures*, vol. 36, no. 4, pp. 317–344, 2010.

[66] A. Malik, A. Girault, and Z. Salcic, "Formal Semantics, Compilation and Execution of the GALS Programming Language DSystemJ," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 7, pp. 1240–1254, jul 2012.

[67] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[68] ITU-T, "Nested Context Language (NCL) and Ginga-NCL," International Telecommunication Union, Geneva, Recommendation H.761, Nov. 2014.

[69] G. F. Lima, R. G. d. A. Azevedo, S. Colcher, and E. H. Haeusler, "Converting NCL Documents to Smix and Fixing Their Semantics and Interpretation in the Process," in *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*, ser. WebMedia '17. New York, NY, USA: ACM, 2017, pp. 109–116.

[70] O. Gaggi and A. Bossi, "Analysis and verification of SMIL documents," *Multimedia Systems*, vol. 17, no. 6, pp. 487–506, nov 2011.

[71] A. Abdelli, "Improving the consistency verification and the quality of multimedia presentations," *International Journal of Critical Computer-Based Systems*, vol. 2, no. 3/4, p. 221, 2011. [Online]. Available: https://doi.org/10.1504/ijccbs.2011.042327

[72] F. Z. Mekahlia, A. Ghomari, S. Yazid, and D. Djenouri, "Temporal and spatial coherence verification in smil documents with hoare logic and disjunctive constraints: A hybrid formal method," *Journal of Integrated Design and Process Science*, vol. 20, no. 3, p. 39–70, Jun 2017. [Online]. Available: http://doi.org/10.3233/jid-2016-0020

[73] W3C, "Scalable Vector Graphics (SVG) 1.1 (Second Edition)," 2011, W3C Recommendation. http://www.w3.org/TR/SVG/.

[74] M. Jourdan, "A formal semantics of SMIL: a web standard to describe multimedia documents," *Computer Standards & Interfaces*, vol. 23, no. 5, pp. 439–455, nov 2001.

[75] A. Bossi and O. Gaggi, "Enriching smil with assertions for temporal validation," in *Proceedings of the 15th ACM International Conference on Multimedia*, ser. MM '07. New York, NY, USA: ACM, 2007, pp. 107–116. [Online]. Available: http://doi.acm.org/10.1145/1291233.1291256

[76] J. dos Santos, C. Braga, and D. C. Muchaluat-Saade, "A rewriting logic semantics for ncl," *Sci. Comput. Program.*, vol. 107, no. C, pp. 64–92, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2015.04.006

[77] ——, "An Executable Semantics for a Multimedia Authoring Language," *Formal Methods: Foundations and Applications. SBMF 2013*, vol. 1, no. 2, pp. 67–82, 2013.

[78] D. Picinin, Jr., J.-M. Farines, and C. Koliver, "An approach to verify live ncl applications," in *Proceedings of the 18th Brazilian Symposium on Multimedia and the Web*, ser. WebMedia '12. New York, NY, USA: ACM, 2012, pp. 223–232. [Online]. Available: http://doi.acm.org/10.1145/2382636.2382685

[79] A. Y. Chang, "An Intelligent Analysis and Verification Model for Consistent SMIL Presentations," *Journal of Convergence Information Technology*, vol. 7, no. 7, pp. 332–341, apr 2012. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84860237444{&}partnerID=tZOtx3y1

[80] D. Picinin, J. M. Farines, C. A. Santos, and C. Koliver, "A design-oriented method to build correct hypermedia documents," *Multimedia Tools and Applications*, pp. 1–30, 2017.

[81] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas, "Safe system-level concurrency on resource-constrained nodes," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517360

[82] G. Berry, "Preemption in concurrent systems," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1993, pp. 72–93. [Online]. Available: https://doi.org/10.1007/3-540-57529-4_44

[83] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358434

[84] S. Buraga and G. Ciobanu, "A rdf-based model for expressing spatio-temporal relations between web sites," in *Proceedings of the Third International Conference on Web Information Systems Engineering, 2002. WISE 2002.*, Dec 2002, pp. 355–361.

[85] S. Laborie, J. Euzenat, and N. Layaïda, "Multimedia document summarization based on a semantic adaptation framework," in *Proceedings of the 2007 International Workshop on Semantically Aware Document Processing and Indexing*, ser. SADPI '07. New York, NY, USA: ACM, 2007, pp. 87–94.

[86] A. Y. Chang, "Design of consistent smil documents for distributed multimedia presentation using temporal algebra," in *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, vol. 1, July 2005, pp. 189–195 Vol. 1.

[87] M. Vazirgiannis, I. Kostalas, and T. Sellis, "Specifying and authoring multimedia scenarios," *IEEE MultiMedia*, vol. 6, no. 3, pp. 24–37, Jul 1999.

[88] D. C. Muchaluat-Saade, R. F. Rodrigues, and L. F. G. Soares, "Xconnector: Extending xlink to provide multimedia synchronization," in *Proceedings of the 2002 ACM Symposium on Document Engineering*, ser. DocEng '02. New York, NY, USA: ACM, 2002, pp. 49–56. [Online]. Available: http://doi.acm.org/10.1145/585058.585069

[89] R. M. d. R. Costa, M. F. Moreno, and L. F. Gomes Soares, "Intermedia synchronization management in dtv systems," in *Proceedings of the Eighth ACM Symposium on Document Engineering*, ser. DocEng '08. New York, NY, USA: ACM, 2008, pp. 289–297.

[90] R. C. Santos, G. F. Lima, F. Sant'Anna, and N. Rodriguez, "CÉU-MEDIA: Local Inter-Media Synchronization Using CÉU," in *Proceedings of the 22Nd Brazilian Symposium on Multimedia and the Web*, ser. Webmedia '16. New York, NY, USA: ACM, 2016, pp. 143–150.

[91] C. d. S. Soares Neto, L. F. G. Soares, and C. S. de Souza, "Tal—template authoring language," *Journal of the Brazilian Computer Society*, vol. 18, no. 3, pp. 185–199, Sep 2012. [Online]. Available: https://doi.org/10.1007/s13173-012-0073-7

[92] J. A. Ferreira dos Santos and D. C. Muchaluat Saade, "Xtemplate 3.0 language: Easing the authoring of ncl programs for interactive digital tv," in *Proceedings of the XV Brazilian Symposium on Multimedia and the Web*, ser. WebMedia '09. New York, NY, USA: ACM, 2009, pp. 17:1–17:8. [Online]. Available: http://doi.acm.org/10.1145/1858477.1858494

[93] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7," Tech. Rep., Jan 2018.

[94] M. Mauve, "How to Keep a Dead Man from Shooting," in *In Proceedings of the 7 th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS) 2000*, 2000, pp. 199–204.

[95] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg, "Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications," vol. 6, no. 1, feb 2004, pp. 47–57.

[96] I. M. Arntzen and F. Daoust. Timing Object. Accessed: 2018-09-03. [Online]. Available: http://webtiming.github.io/timingobject/

[97] W3C. Media and Entertainment Interest Group. https://www.w3.org/2011/webtv/.

[98] D. Geerts, I. Vaishnavi, R. Mekuria, O. van Deventer, and P. Cesar, "Are we in sync?: Synchronization requirements for watching online video together." in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 311–314.

[99] F. Boronat, M. Montagud, and V. Vidal, "Master Selection Policies for Inter-destination Multimedia Synchronization in Distributed Applications," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, jul 2011, pp. 269–277.

[100] K.-d. Seo, T.-j. Jung, J. Yoo, C. K. Kim, and J. Hong, "A new timing model design for MPEG Media Transport (MMT)," in *IEEE international Symposium on Broadband Multimedia Systems and Broadcasting*. IEEE, jun 2012, pp. 1–5.

[101] M. Montagud, F. Boronat, H. Stokking, and P. Cesar, "Design, development and assessment of control schemes for IDMS in a standardized RTCP-based solution," *Computer Networks*, vol. 70, pp. 240–259, sep 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128614002278

[102] V. Vinayagamoorthy, R. Ramdhany, and M. Hammond, "Enabling Frame-Accurate Synchronised Companion Screen Experiences," in *Proceedings of the ACM International Conference on Interactive Experiences for TV and Online Video - TVX '16*. New York, New York, USA: ACM Press, 2016, pp. 83–92. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2932206.2932214

[103] S. U. Din and D. Bulterman, "Synchronization techniques in distributed multimedia presentation," in *MMEDIA - International Conferences on Advances in Multimedia*, 2012, pp. 1–9. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84893310632{&}partnerID=tZOtx3y1

[104] B. Rainer, S. Petscharnig, and C. Timmerer, "Merge and forward: self-organized inter-destination multimedia synchronization," in *Proceedings of the 6th ACM Multimedia Systems Conference on - MMSys '15*. ACM Press, mar 2015, pp. 77–80.

[105] C. Köhnen, N. Hellhund, J. Renz, and J. Müller, "Inter-Device and Inter-Media Synchronization in HBB-NEXT," in *Proceedings of the Media Synchronization Workshop (MediaSync 2013)*, 2013, pp. 1–4.

[106] H. Stokking, M. van Deventer, O. Niamut, F. Walraven, and R. Mekuria, "IPTV inter-destination synchronization: A network-based approach," in *2010 14th International Conference on Intelligence in Next Generation Networks*. IEEE, oct 2010, pp. 1–6.

[107] R. Bardeli, J. Schwenninger, and D. Stein, "Audio Fingerprinting for Media Synchronisation and Duplicate Detection," in *Proceedings of the Media Synchronization Workshop (MEDIASYNC 2012)*, Berlin, Germany, 2012.

[108] N. Q. K. Duong, C. Howson, and Y. Legallais, "Fast second screen tv synchronization combining audio fingerprint technique and generalized cross correlation," in *2012 IEEE Second International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, Sept 2012, pp. 241–244.

[109] L. C. Villa Real, R. Laiola Guimarães, and P. Avegliano, "Dynamic adjustment of subtitles using audio fingerprints," in *Proceedings of the 23rd ACM International Conference on Multimedia*, ser. MM '15. New York, NY, USA: ACM, 2015, pp. 975–978. [Online]. Available: http://doi.acm.org/10.1145/2733373.2806378

[110] M. Montagud and F. Boronat, "On the Use of Adaptive Media Playout for Inter-Destination Synchronization," *IEEE Communications Letters*, vol. 15, no. 8, pp. 863–865, aug 2011.

[111] B. Rainer and C. Timmerer, "Adaptive Media Playout for Inter-Destination Media Synchronization," in *2013 Fifth International Workshop on Quality of Multimedia Experience (QoMEX)*. IEEE, jul 2013, pp. 44–45.

[112] ——, "Self-Organized Inter-Destination Multimedia Synchronization For Adaptive Media Streaming," in *Proceedings of the ACM International Conference on Multimedia - MM '14*. New York, New York, USA: ACM Press, nov 2014, pp. 327–336.

[113] ATSC, "ATSC Standard: Companion Device (A/338)," Washington, D.C., USA, Tech. Rep., 2017.

[114] ETSI TS 103 286-1, "Digital Video Broadcasting (DVB); Companion Screens and Streams; Part 1: Concepts, Roles and Overall Architecture," Sophia Antipolis,France, Tech. Rep., 2015.

[115] M. Nebeling, M. Grossniklaus, S. Leone, and M. C. Norrie, "XCML: providing context-aware language extensions for the specification of multi-device web applications," *World Wide Web*, vol. 15, no. 4, pp. 447–481, jul 2012. [Online]. Available: http://link.springer.com/10.1007/s11280-011-0152-2

[116] J. Nichols and B. A. Myers, "Creating a lightweight user interface description language," *ACM Transactions on Computer-Human Interaction*, vol. 16, no. 4, pp. 1–37, nov 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1614390.1614392

[117] F. Paterno', C. Santoro, and L. D. Spano, "MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," *ACM Transactions on Computer-Human Interaction*, vol. 16, no. 4, pp. 1–30, nov 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1614390.1614394

[118] S. K. Badam and N. Elmqvist, "PolyChrome: A Cross-Device Framework for Collaborative Web Visualization," in *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces - ITS '14*. New York, New York, USA: ACM Press, 2014, pp. 109–118. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2669485.2669518

[119] P.-Y. P. Chi and Y. Li, "Weave: Scripting Cross-Device Wearable Interaction," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*. New York, New York, USA: ACM Press, 2015, pp. 3923–3932. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2702123.2702451

[120] B. Hartmann, M. Beaudouin-Lafon, and W. E. Mackay, "HydraScope: Creating Multi-Surface Meta-Applications Through View Synchronization and Input Multiplexing," in *Proceedings of the 2nd ACM*

*International Symposium on Pervasive Displays - PerDis '13*. New York, New York, USA: ACM Press, 2013, p. 43. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.309.8136http://dl.acm.org/citation.cfm?doid=2491568.2491578

[121] S. Houben and N. Marquardt, "WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*. New York, New York, USA: ACM Press, 2015, pp. 1247–1256. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2702123.2702215

[122] M. Nebeling, M. Husmann, C. Zimmerli, G. Valente, and M. C. Norrie, "XDSession: integrated development and testing of cross-device applications," in *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '15*. New York, New York, USA: ACM Press, 2015, pp. 22–27. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2774225.2775075

[123] P. Hamilton, D. J. Wigdor, P. Hamilton, and D. J. Wigdor, "Conductor: enabling and understanding cross-device interaction," in *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. New York, New York, USA: ACM Press, 2014, pp. 2773–2782. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2556288.2557170

[124] G. Ghiani, F. Paternò, and C. Santoro, "Push and pull of web user interfaces in multi-device environments," in *Proceedings of the International Working Conference on Advanced Visual Interfaces - AVI '12*. New York, New York, USA: ACM Press, 2012, p. 10. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2254556.2254563

[125] L. Frosini and F. Paternò, "User interface distribution in multi-device and multi-user environments with dynamically migrating engines," in *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems - EICS '14*. New York, New York, USA: ACM Press, 2014, pp. 55–64. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2607023.2607032

[126] J. Yang and D. Wigdor, "Panelrama: enabling easy specification of cross-device web applications," in *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. New York, New York, USA: ACM Press, 2014, pp. 2783–2792. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2556288.2557199

[127] M. Sarkis, C. Concolato, and J.-C. Dufourd, "A multi-screen refactoring system for video-centric web applications," *Multimedia Tools and Applications*, jan 2017. [Online]. Available: http://link.springer.com/10.1007/s11042-017-4357-y

[128] A. Gallidabino and C. Pautasso, "The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices," in *Proceedings of the 25th International Conference Companion on World Wide Web - WWW '16 Companion*. New York, New York, USA: ACM Press, 2016, pp. 183–186. [Online]. Available: http://dx.doi.org/10.1145/2872518.2890538.http://dl.acm.org/citation.cfm?doid=2872518.2890538

[129] D. Kovachev, D. Renzel, P. Nicolaescu, and R. Klamma, "DireWolf - Distributing and Migrating User Interfaces for Widget-Based Web Applications." Springer, Berlin, Heidelberg, 2013, pp. 99–113. [Online]. Available: http://link.springer.com/10.1007/978-3-642-39200-9{_}10

[130] M. Ciampi, "Design of a Layered Component-Based System for Sharing Visualization Objects," *International Journal of Advanced Computer Engineering*, vol. 5, no. 1, 2012.

[131] P. Lange, R. Weller, and G. Zachmann, "Scalable concurrency control for massively collaborative virtual environments," in *Proceedings of the 7th ACM International Workshop on Massively Multiuser Virtual Environments - MMVE '15*. New York, New York, USA: ACM Press, 2015, pp. 7–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2723695.2723699

[132] D. Roberts and R. Wolff, "Controlling consistency within collaborative virtual environments," *Proceedings - Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT 2004*, pp. 46–51, 2004.

[133] A. Boukerche, N. McGraw, and R. Araujo, "A Grid-Filtered Region-Based Approach to Support Synchronization in Large-Scale Distributed Interactive Virtual Environments," in *International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE, 2005, pp. 525–530. [Online]. Available: http://ieeexplore.ieee.org/document/1488738/

[134] "IEEE Standard for Distributed Interactive Simulation - Application Protocols," *IEEE Std. 1278.1-1995*, 1995.

[135] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the Internet," *IEEE Network*, vol. 13, no. 4, pp. 6–15, 1999. [Online]. Available: http://ieeexplore.ieee.org/document/777437/

[136] P. Bettner and M. Terrano, "500 Archers on a 28.8: Network Programming in Age of Empires and Beyond Mark Terrano," in *Game Developers Conference*, 2001. [Online]. Available: https://zoo.cs.yale.edu/classes/cs538/readings/papers/terrano{_}1500arch.pdf

[137] Y. W. Bernier, "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization," in *Game Developers Conference*, 2001. [Online]. Available: https://www.gamedevs.org/uploads/latency-compensation-in-client-server-protocols.pdf

[138] J. Smed and H. Hakonen, *Algorithms and networking for computer games*, 2nd ed. Wiley, 2006.

[139] P. Sharkey, M. Ryan, and D. Roberts, "A local perception filter for distributed virtual environments," in *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium*. IEEE Comput. Soc, 1988, pp. 242–249.